Simit: A Language for Physical Simulation

FREDRIK KJOLSTAD
Massachusetts Institute of Technology
SHOAIB KAMIL
Adobe
JONATHAN RAGAN-KELLEY
Stanford University
DAVID I.W. LEVIN
Disney Research
SHINJIRO SUEDA
California Polytechnic State University
DESAI CHEN
Massachusetts Institute of Technology
ETIENNE VOUGA
University of Texas at Austin
DANNY M. KAUFMAN
Adobe
and
GURTEJ KANWAR, WOJCIECH MATUSIK, and SAMAN AMARASINGHE
Massachusetts Institute of Technology

Using existing programming tools, writing high-performance simulation code is labor intensive and requires sacrificing readability and portability. The alternative is to prototype simulations in a high-level language like Matlab, thereby sacrificing performance. The Matlab programming model naturally describes the behavior of an entire physical system using the language of linear algebra. However, simulations also manipulate individual geometric elements, which are best represented using linked data structures like meshes. Translating between the linked data structures and linear algebra comes at significant cost, both to the programmer and the machine. High-performance implementations avoid the cost by rephrasing the computation in terms of linked or index data structures, leaving the code complicated and monolithic, often increasing its size by an order of magnitude.

In this paper, we present Simit, a new language for physical simulations that lets the programmer view the system both as a linked data structure in the form of a hypergraph, and as a set of global vectors, matrices and tensors depending on what is convenient at any given time. Simit provides a novel assembly construct that makes it conceptually easy and computationally efficient to move between the two abstractions. Using the information provided by the assembly construct, the compiler generates efficient in-place computation on the graph. We demonstrate that Simit is easy to use: a Simit program is typically shorter than a Matlab program; that it is high-performance: a Simit program running sequentially on a CPU performs comparably to hand-optimized simulations; and that it is portable: Simit programs can be compiled for GPUs with no change to the program, delivering 4-20x speedups over our optimized CPU code.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*

Additional Key Words and Phrases: Graph, Matrix, Tensor, Simulation

DOI: XXXX

1. INTRODUCTION

Efficient computer simulations of physical phenomena are notoriously difficult to engineer, requiring careful optimization to achieve good performance. This stands in stark contrast to the elegance of the underlying physical laws; for example, the behavior of an elastic object, modeled (for ease of exposition) as a network of masses connected by springs, is determined by a single quadratic equation, Hooke's law, applied homogeneously to every spring in the network. While Hooke's law describes the local behavior of the mass-spring network, it tells us relatively little about its global, emergent behavior. This global behavior, such as how an entire object will deform, is also described by simple but coupled systems of equations.

Each of these two aspects of the physical system—its local interactions and global evolution laws—admit different useful abstractions. The local behavior of the system can be naturally encoded in a graph, with the degrees of freedom stored on vertices, and interactions between degrees of freedom represented as edges. These interactions are described by local physical laws (like Hooke's law from above), applied uniformly, like a stencil, over all of the edges. However, this stencil interpretation is ill-suited for representing the coupled equations which describe global behaviors. Once discretized and linearized, these global operations are most naturally expressed in the

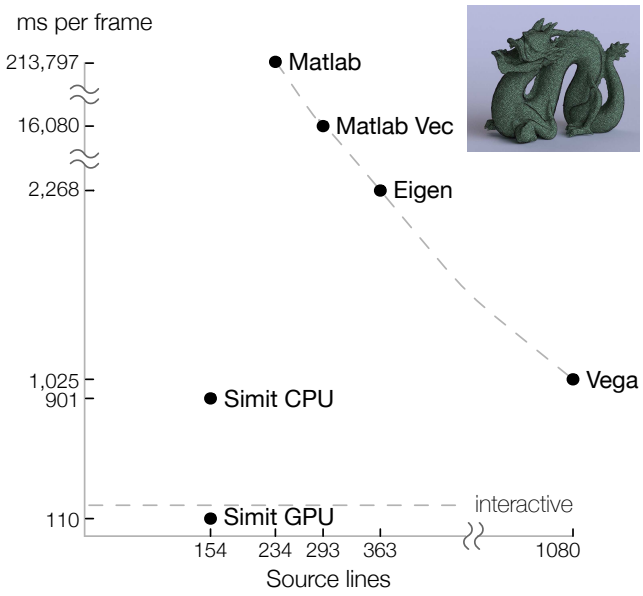


Fig. 1: Scatter plot that shows the relationship between the code size and runtime of a Neo-Hookean FEM simulation implemented using (Vectorized) Matlab, the optimized Eigen Linear Algebra library, the hand-optimized Vega FEM framework, and Simit. The runtimes are for a dragon with 160,743 tetrahedral elements. The trend is that you get more performance by writing more code, however, with Simit you get both performance and productivity. Simit requires fewer lines of code than the Matlab implementation and runs faster than the hand-optimized Vega library on a single-threaded CPU. On a GPU, the Simit implementation runs 6× faster with no code changes.

language of linear algebra, where all of the system data is aggregated into huge but sparse matrices and vectors.

The easiest way for a programmer to reason about a physical simulation, and hence a common idiom when implementing one, is to swap back and forth between the global and local abstractions. First, a graph or mesh library might be used to store a mass spring system. Local forces and force Jacobians are computed with uniform, local stencils on the mesh and copied into large sparse matrices and vectors, which are then handed off to optimized sparse linear algebra libraries to calculate the updated global state of the simulation. Finally this updated state is copied back onto the mesh.

While straightforward to conceptualize, the strategy of copying data back and forth between the graph and matrix representations incurs high performance costs as a result of data translation, and the inability to optimize globally across linear algebra operations. To overcome this inefficiency, highly-optimized simulations, like those used for games and other real-time applications, are built as monolithic codes that perform assembly and linear algebra on a single set of data structures, often by computing in-place on the mesh. Building such a monolithic code requires enormous programmer effort and expertise. Doing so while keeping the system maintainable and extensible, or allowing retargeting of the same code to multiple architectures such as GPUs and CPUs, is nearly impossible.

The Simit Language

To let users take advantage of implicit local-global structure without the performance pitfalls described above, we propose a new programming language called Simit that *natively supports switching between graph and matrix views* of the simulation. Because Simit is aware of the local-global duality at the language level, it lets simulation codes be concise, fast (see Figure 1), and portable (compiling to both CPU and GPU with no source code change). Simit makes use of three key abstractions: first, the local view is defined using a hypergraph data structure, where nodes represent degrees of freedom and hyperedges relationships such as force stencils, finite elements, and joints. Hyperedges are used instead of regular edges to support relationships between more than two vertices. Second, the local operations to be performed are encoded as functions acting on neighborhoods of the graph (such as Hooke's law). Lastly, and most importantly, the user specifies how global vectors and matrices are related to the hypergraph and local functions. For instance, the user might specify that the global force vector is to be built by applying Hooke's law to each spring and summing the forces acting on each mass. The key point is that defining a global matrix in this way is *not* an imperative instruction for Simit to materialize a matrix in memory: rather, it is an abstract definition of the matrix (much as one would define the matrix in a mathematical paper). The programmer can then operate on that abstract matrix using linear algebra operations; Simit analyzes these operations and translates them into operations on the hypergraph. Because Simit understands the mapping from the graph to the matrix, it can globally optimize the code it generates while still allowing the programmer to reason about the simulation in the most natural way: as both local graph operations and linear algebra on sparse matrices.

Simit's performance comes from its design and is made possible by Simit's careful choice of abstractions. Three features (Section 9) come together to yield the surprising performance shown in Figure 1:

In-place Computation is made possible by the tensor assembly construct that lets the compiler understand the relationship between global operations and the graph and turn global linear algebra into in-place local operations on the graph structure. This means that Simit does not need to generate sparse matrix index structures or allocate matrix and vector memory at runtime;

Index Expression Fusion is used to fuse linear algebra operations, yielding loops that perform multiple operations at once. Further, due to in-place computation even sparse operations can be fused; and

Simit's Type System with natively blocked vectors, matrices and tensors, lets it perform efficient dense block computation by emitting dense loops as sub-computations of sparse operations.

Simit's performance can be enhanced even further by emitting vector instructions or providing multi-threaded CPU execution, optimizations that are planned for a future version of the compiler.

Scope

Simit is designed for algorithms where local stencils are applied to a graph of fixed topology to form large, global matrices and vectors, to which numerical algorithms are applied and the results written back onto the graph. This abstraction perfectly fits many physical simulation problems such as mass-spring networks (where hyperedges are the springs), cloth (the bending and stretching force stencils), viscoelastic deformable bodies (the finite elements), etc. At this time the Simit language does not natively support changing the graph topology (such as occurs with fracture or penalty-based impact forces) or simulation components that do not fit the graph abstraction (such as collision detection spatial data structures, semi-Lagrangian advection of fluids). However, as discussed in Section 5, Simit is interoperable with C++ code and libraries, which can be used to circumvent some of these limitations. For example, the graph topology can be changed using Simit's C++ library between timesteps, and Simit will recompute necessary indices.

The target audience for Simit is researchers, practitioners, and educators who want to develop physical simulation code that is more readable, maintainable, and retargetable than Matlab or C++, while also being significantly more efficient (comparable to optimized physics libraries like SOFA). Simit programs will not outperform hand-tuned CUDA code, or be simpler to use than problem-specific tools like FreeFem++, but Simit occupies a sweet spot balancing these goals (see Figure 1 and benchmarks in Section 8) that is ideal for a general-purpose physical simulation language.

Contributions

Simit is the first system that allows the development of physics code that is simultaneously:

Concise The Simit language has Matlab-like syntax that lets algorithms be implemented in a compact, readable form that closely mirrors their mathematical expression. In addition, Simit matrices assembled from hypergraphs are indexed by hypergraph elements like vertices and edges rather than by raw integers, significantly simplifying indexing code and eliminating bugs.

Expressive The Simit language consists of linear algebra operations augmented with control flow that let developers implement a wide range of algorithms ranging from finite elements for deformable bodies, to cloth simulations and more. Moreover, the powerful hypergraph abstraction allows easy specification of complex geometric data structures.

Fast The Simit compiler produces high-performance executable code comparable to that of hand-optimized end-to-end libraries and tools, as validated against the state-of-the-art SOFA [Faure et al. 2007] and Vega [Sin et al. 2013] real-time simulation frameworks. Simulations can now be written as easily as a traditional prototype and yet run as fast as a high performance implementation without manual optimization.

Performance Portable A Simit program can be compiled to both CPUs and GPUs with no additional programmer effort, while generating efficient code for each architecture. Where Simit delivers performance comparable to hand-optimized CPU code on the same processor, the same simple Simit program delivers roughly an order of magnitude higher performance on a modern GPU in our benchmarks, with no changes to the program.

Interoperable Simit hypergraphs and program execution are exposed as C++ APIs, so developers can seamlessly integrate with existing C++ programs, algorithms and libraries.

2. RELATED WORK

The Simit programming model draws on ideas from programming systems, numerical and simulation libraries, and physical and mathematical frameworks.

Hand-Optimized Physical Simulations

Researchers have explored many techniques to optimize simulation problems for CPUs and GPUs. These codes tend to be memory bound so much of this work has gone into optimizing data layout and data access patterns. For implicit codes, most of the time is spent assembling and solving sparse systems of linear equations, and the most important consideration is the choice of solver algorithm and implementation. Popular solvers are the Conjugate Gradient method (CG), e.g. [Faure et al. 2007]; Projected CG (PCG), e.g. [Weber et al. 2013]; and Multigrid (MG), e.g. [McAdams et al. 2011; Dick et al. 2011]. Since these solvers are iterative most of the time is spent in sparse matrix-vector multiplications (SpMV) or equivalent operation, and an efficient sparse matrix representation is essential.

SOFA [2007] encapsulates SpMV operations used in iterative solves as the application of a force field. The SOFA mass-spring force field is completely matrix-free; no part of the global sparse matrix is stored and the matrix blocks are computed on demand in the SpMV. The SOFA FEM force field is only partly matrix-free. Because it is expensive to compute FEM element stiffness matrices, they should not be recomputed each CG iteration. The FEM therefore consists of an assembly stage that computes element stiffness matrices, and stores them on the elements. The global stiffness matrix is thus stored in partly assembled form, prior to the summation of the element stiffness matrices that affect each vertex.

The Hexahedral Corotational FEM implementation of Dick et al. [2011] goes one step further and stores element stiffness matrices in their final assembled form on the mesh vertices. This is equivalent to storing them in a BCSR format, where each matrix row is stored on the corresponding vertex, but without the need to store an explicit matrix index since the mesh is effectively the matrix's index.

In the work of Weber et al. [2013] they explore a new general matrix storage format they call BIN-CSR. Unlike the (partly) matrix-free approaches above, they store matrix indices instead of using the mesh data structure. To speed up GPU execution they divide matrix rows into bins that store data using the ELLPACK format [1989].

The current Simit compiler produces code that uses the same storage scheme as Dick et al. for global matrices. That is, matrix blocks are stored in their final assembled form on the vertices. Further, similar to the approach of Dick et al., compiled Simit code uses graph neighbor indices as the index of all non-diagonal assembled matrices (diagonal matrices do not need indices). However, Simit is a high-level language where the user does not have to implement these optimization choices; rather the compiler takes care of it. A Simit implementation of the Dick et al. application does not yet get as good performance as their hand-optimized implementation, but the user writes an order of magnitude less code and gets much better performance than a naive C++ implementation. Further, the code is high-level linear algebra physics code, rather than optimized C++ and CUDA kernels. Finally, Simit can compile the same application to multiple architectures, such as CPUs and GPUs, and as new architectures such as the Xeon Phi become available, the Simit compiler can target them thus removing the need to reimplement Simit applications. See Section 8.8 for more details.

Direct solvers are an important alternative to iterative solvers that often perform better [Botsch et al. 2005]. However, in this work we focus on iterative solvers due to their low memory consumption and popularity for physical simulation. We leave support for implementing direct solvers in Simit as future work.

Libraries for Physical Simulation

A wide range of libraries for the physical simulation of deformable bodies with varying degrees of generality are available [Pommier and Renard 2005; Faure et al. 2007; Dubey et al. 2011; Sin et al. 2013; Comsol 2005; Hibbett et al. 1998; Kohnke 1999], while still others specifically target rigid and multi-body systems with domain specific custom optimizations [Coumans et al. 2006; Smith et al. 2005; Liu 2014]. These simulation codes are broad and many serve double duty as both production codes and algorithmic testbeds. As such they often provide collections of algorithms rather than customizations suited to a particular timestepping and/or spatial discretization model. With broad scope comes convenience but even so inter-library communication is often hampered by data conversion while generality often limits the degree of optimization.

Mesh Data Structures

Simulation codes often use third-party libraries that support higher-level manipulation of the simulation mesh. A *half-edge data structure* [Eastman and Weiss 1982] (from, e.g., the OpenMesh library [Botsch et al. 2002]) is one popular method for describing a mesh which allows efficient connectivity queries and neighborhood circulation. Alternatives that target different application requirements (manifold vs. nonmanifold, oriented vs unoriented, etc.) abound, such as winged-edge [Baumgart 1972] or quad-edge [Guibas and Stolfi 1985] data structures. Modern software packages like CGAL [CGAL 2015] provide sophisticated tools on top of many of these data structures for performing common geometry operations. Simit's hierarchical hyper-edges provide sufficient expressiveness to let users build semantically rich data structures like meshes, while not limiting the user to meshes.

DSLs for Computer Graphics

Graphics has a long history of using domain-specific languages and abstractions to provide high performance, and performance portability, from relatively simple code. Most visible are shading languages and the graphics pipeline [Hanrahan and Lawson 1990; Segal and Akeley 1994; Mark et al. 2003; Blythe 2006; Parker et al. 2010]. Image processing languages also have a long history [Holzmann 1988; Elliott 2001; Ragan-Kelley et al. 2012], and more recently domain-specific languages have been proposed for new domains like 3D printing [Vidimče et al. 2013]. In physical simulation, Guenter et al. built the $D*$ system for symbolic differentiation, and demonstrated its application to modeling and simulation [Guenter and Lee 2009]. $D*$ is an elegant abstraction, but its implementation focuses less on optimized simulation performance, and its model cannot express features important to many of our motivating applications.

Graph Programming Models

A number of programming systems address computation over graphs or graph-like data structures, including GraphLab [Low et al. 2010], Galois [Pingali et al. 2011], Liszt [DeVito et al. 2011], SociaLite [Jiwon Seo 2013], and GreenMarl [Sungpack Hong and Olukotun 2012]. In these systems, programs are generally written as explicit in-place computations using stencils on the graph, providing a much lower level of abstraction than linear algebra over whole systems. Of these, GraphLab and SociaLite focus on distributed systems, while we currently focus on single-node/shared memory execution. SociaLite and GreenMarl focus on scaling traditional graph algorithms (e.g., breadth-first search and betweenness centrality) to large graphs. Liszt exposes a programming model over meshes. Computations are written in an imperative fashion, but must look like stencils, so it only allows element-wise operations and reductions. This is similar to the programming model used for assembly in Simit, but it has no corollary to Simit's linear algebra for easy operation on whole systems. Galois exposes explicit in-place programming via a similarly low-level, but very dynamic programming model, which inhibits compiler analysis and optimization.

Programming Systems for Linear Algebra

Our linear algebra syntax is inspired by Matlab [2014], the most successful high-productivity tool in this domain. However, the tensor assembly map operator, together with coordinate-free indexing and hierarchically blocked tensors, dramatically reduces indexing complexity during matrix assembly, and exposes structure critical to our compiler optimizations. Eigen [Guennebaud et al. 2010] is a C++ library for linear algebra which uses aggressive template metaprogramming to specialize and optimize linear algebra computations at compile time, including fusion of multiple operations and vectorization. It does an impressive job exposing linear algebra operations to C++, and aggressive vectorization delivers impressive inner-loop performance, but assembly is still both challenging for programmers and computationally expensive during execution.

3. FINITE ELEMENT METHOD EXAMPLE

To make things concrete, we start by discussing an example of a paradigmatic Simit program: a Finite Element Method (FEM) statics simulation that uses Newton's method to compute the final configuration of a deforming object. Figure 2 shows the source code for this example. The implementation of `compute_tet_stiffness` and `compute_tet_force` depends on the material model chosen by the user and are omitted. In this section we introduce Simit concepts with respect to the example, but we will come back to them in Section 4 with rigorous definitions.

As is typical, this Simit application consists of five parts: (1) graph definitions, (2) functions that are applied to each graph vertex or edge to compute new values based on neighbors, (3) functions that compute local contributions of vertices and edges to global vectors and matrices, (4) assemblies that aggregate the local contributions into global vectors and matrices, and (5) code that computes with vectors and matrices.

Step 1 is to define a graph data structure. Graphs consist of elements (objects) that are organized in vertex sets and edge sets. Lines 1–16 define a Simit graph where edges are tetrahedra and vertices their degrees of freedom. Lines 1–5 define an element of type `Vertex` that represents a tetrahedron's degrees of freedom. It has three fields: a coordinate `x`, a velocity `v`, and an external force `fe`. Next, lines 7–12 define a `Tet` element that represents an FEM tetrahedron with four fields: shear modulus `u`, Lame's first parameter `l`, volume `W`, and the strain-displacement 3×3 matrix `B`. Finally, lines 15–16 define a vertex set `verts` with `Vertex` elements, and a edge set `tets` with `Tet` elements. Since `tets` is an edge set, its definition lists the sets containing the edge endpoints; a tetrahedron connects four vertices (see Figure 3). Simit graphs are hypergraphs, which means that edges can connect any fixed number of vertices.

Step 2 is to define and apply a function `precompute_vol` to precompute the the volume of every tetrahedron. In Simit this can be done by defining the *stencil function* `precompute_vol` shown on lines 19–22. Simit stencil functions are similar to the update functions of data-graph libraries such as GraphLab [Low et al. 2010] and can be applied to every element of a set (`tets`) and its endpoints (`verts`). Stencil functions define one or more `inout` parameters that have pass-by-reference semantics and that can be modified. Lines 24–26 shows the Simit function `init` that can be called from C++ to precompute volumes. The exported function contains a single statement that applies the stencil function to every tetrahedron in `tets`.

Step 3 defines functions that compute the local contributions of a vertex or an edge to global vectors and matrices. Lines 29–34 define `tet_force` which computes the forces exerted by a tetrahedron on its vertices. The function takes two arguments, the tetrahedron and a tuple containing its vertices. It returns a global vector `f` that contains the local force contributions of the tetrahedron. Line 32 computes the tetrahedron forces and assigns them to `f`. Since `tet_force` only has access to the vertices of one tetrahedron, it can only write to four locations in the global vector. This is sufficient, however, since a tetrahedron only *directly* influences its own vertices.

```
1  element Vertex
2    x  : vector[3](float);   % position
3    v  : vector[3](float);   % velocity
4    fe : vector[3](float);   % external force
5  end
6
7  element Tet
8    u : float;               % shear modulus
9    l : float;               % Lame's first parameter
10   W : float;               % volume
11   B : matrix[3,3](float);  % strain-displacement
12 end
13
14 % graph vertices and (tetrahedron) hyperedges
15 extern verts : set{Vertex};
16 extern tets  : set{Tet}(verts, verts, verts, verts);
17
18 % precompute tetrahedron volume
19 func precompute_vol(inout t : Tet, v : (Vertex*4))
20   t.B = compute_B(v);
21   t.W = -det(B)/6.0;
22 end
23
24 export func init
25   apply precompute_vol to tets;
26 end
27
28 % computes the force of a tetrahedron on its vertices
29 func tet_force(t : Tet, v : (Vertex*4))
30     -> f : vector[verts](vector[3](float))
31   for i in 0:4
32     f(v(i)) = compute_tet_force(t,v,i);
33   end
34 end
35
36 % computes the stiffness of a tetrahedron
37 func tet_stiffness(t : Tet, v : (Vertex*4))
38     -> K : matrix[verts,verts](matrix[3,3](float))
39   for i in 0:4
40     for j in 0:4
41       K(v(i),v(j)) = compute_tet_stiffness(t,v,i,j);
42     end
43   end
44 end
45
46 % newton's method timestepper
47 export func newton_method
48   tol = 1e-6;
49   while abs(f - verts.fe) > tol
50     f = map tet_force to tets reduce +;
51     K = map tet_stiffness to tets reduce +;
52
53     verts.x = verts.x + K\(verts.fe - f);
54   end
55 end
```

1 2 3 4 5

Fig. 2: Simit code for a Finite Element Method (FEM) simulation of statics. The code contains: (1) graph definitions, (2) a function that is applied to each tetrahedron to precompute it's volume, (3) functions that compute local contributions of each tetrahedra to global vectors and matrices, (4) assemblies that aggregate the local contributions into global vectors and matrices, and (5) code that computes with those vectors and matrices. The `compute_*` functions have been omitted for brevity. For more Simit code, see Appendix A for a full Finite Element Dynamics Simulation.

Step 4 uses a Simit assembly map to aggregate the local contributions computed from vertices and edges in the previous stage into a global vector and matrix. Line 50 assembles a global force vector by summing the local force contributions computed by applying `tet_force` to every tetrahedron. The result is a dense force vector `f` that contains the force of every tetrahedron on its vertices.

Finally, Step 5 is to compute with the assembled global vector and matrix. The results of these computations are typically vectors that

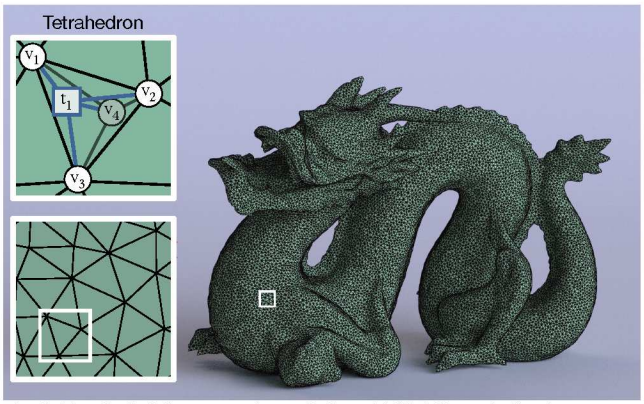


Fig. 3: Tetrahedral dragon mesh consisting of 160,743 tetrahedra that connect 46,779 vertices. Windows zoom in on a region on the mesh and a single tetrahedron. The tetrahedron is modeled as a `Tet` edge $t_1 = \{v_1, v_2, v_3, v_4\}$.

are stored to fields of graph vertices or edges. Line 53 reads the `x` position field from the `verts` set, computes a new position and writes that position back to `verts.x`. Reading a field from a set results in a global vector whose blocks are the fields of the set's elements. Writing a global vector to a set field works the opposite way: the vector's blocks are written to the set elements. In this example the computation uses the linear solve \ operator to perform a linearly implicit time-step, but many other approaches are possible.

4. PROGRAMMING MODEL

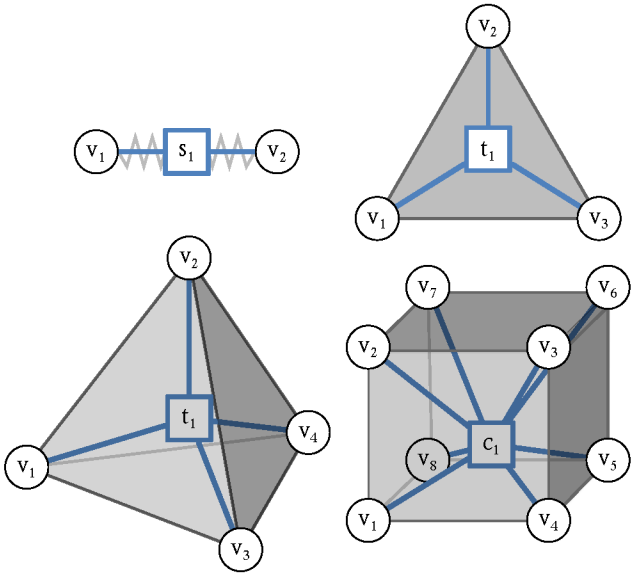
Simit's programming model is designed around the observation that a physical system is typically graph structured, while computation on the system is best expressed as global linear and multi-linear algebra. Thus, the Simit **data model** consists of two abstract data structures: **hypergraphs** and **tensors**. Hypergraphs generalize graphs by letting edges connect any n-element subset of vertices instead of just pairs. Tensors generalize scalars, vectors and matrices, that respectively are indexed by 0, 1 and 2 indices, to an arbitrary number of indices.

We also describe two new operations on hypergraphs and tensors: **tensor assemblies**, and **index expressions**. Tensor assemblies map graphs to tensors, while index expressions compute with tensors.

4.1 Hypergraphs with Hierarchical Edges

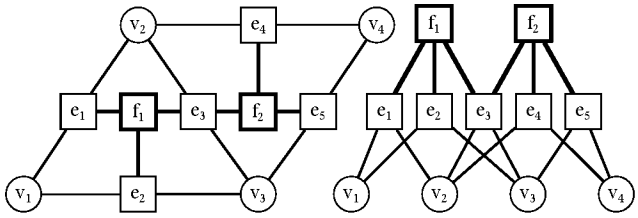
Hypergraphs are ordered pairs $H = (V, E)$, comprising a set V of vertices and a set E of hyperedges that are n-element subsets of V. The number of elements a hyperedge connects is its **cardinality**, and we call a hyperedge of cardinality n an **n-edge**. Thus, hypergraphs generalize graphs where edges must have a cardinality of two. Hypergraphs are useful for describing relationships between vertices that are more complex than binary relationships. Figure 4 shows four examples of hyperedges: a 2-edge, a 3-edge, a 4-edge and an 8-edge (in blue) are used to represent a spring, a geometric triangle, a tetrahedron and a hexahedron (in grey). Although these hyperedges are used to model geometric mesh relationships, hyperedges can be used to model any relationship. For example, a 2-edge can also be used to represent a joint between two rigid bodies or the relationship between two neurons, and a 3-edge can represent a clause in a 3-SAT instance.

Simit hypergraphs generalize normal hypergraphs and are ordered tuples $H_{Simit} = (S_1, ..., S_m)$, where S_i is the ith set whose elements connect 0 or n elements from other sets, called its **endpoints**. We refer to a set with cardinality 0 as a **vertex set** and a set of



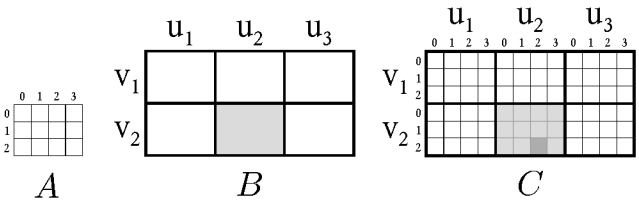

```
springs : set{Spring}(verts,verts);
trigs   : set{Triangle}(verts,verts,verts);
tets    : set{Tetrahedron}(verts,verts,verts,verts);
hexes   : set{Hexahedron}(verts,verts,verts,verts,
                          verts,verts,verts,verts);
```

Fig. 4: Four geometric elements are shown in gray: a spring, a triangle, a tetrahedra and a cube. Vertices are shown as black circles. Simit edges of cardinality three, four and eight are shown as blue squares.




```
verts : set{Vertex};
edges : set{Edge}(verts,verts);
faces : set{Face}(edges,edges,edges);
```

Fig. 5: Hierarchical hyperedges model triangles with faces, edges, and vertices. On the left two triangles with faces f_1 and f_2 are laid flat. On the right the same triangles are arranged to show the topological hierarchy.




```
A : matrix[3,4](float)
B : matrix[V,U](float)
C : matrix[V,U](matrix[3,4](float))
```

Fig. 6: Three Simit matrices. On left is a basic 3×4 matrix A. In the middle is a matrix B whose dimensions are the sets V and U. The matrix is indexed by pairs of elements from U, V, e.g. $B(\mathrm{v}_2, \mathrm{u}_2)$. Finally, on the right is a blocked matrix C with $V \times U$ blocks of size 3×4. A block of this matrix is located by a pair of elements fro V, U, e.g. $C(\mathrm{v}_2, \mathrm{u}_2)$, and an element can be indexed using a a pair of indices per matrix hierarchy, e.g. $C(\mathrm{v}_2, \mathrm{u}_2)(2, 2)$.

cardinality n greater than 0 as an **edge set**. Thus, Simit hypergraphs can have any number of vertex and edge sets, and edge sets can have one or more endpoints. The endpoints of an n-cardinality edge set are a set relation over n other vertex or edge sets. That is, each endpoint of an edge is an element from the corresponding endpoint set. This means that edge sets can connect multiple distinct sets, and we call such edge sets **heterogeneous edge sets**.

Elements (vertices and edges) of hypergraph sets can contain data. An element's data is a tuple whose entries are called **fields**. This is equivalent to record or struct types in other languages. Fields can be scalars, vectors, matrices or tensors. For example, the `Vertex` element on lines 1–5 in Figure 2 has three vector fields `x`, `v` and `fe`. We say that a hypergraph set has the same fields as its elements; however, a field of a set is a vector whose blocks are the fields of the set's elements. Blocked vector types are described in Section 4.2.

Edge sets can connect other edge sets, so we say that Simit supports **hierarchical edges**. Hierarchical edges have important applications in physical simulation because they let us represent the topology of a mesh. Figure 5 demonstrates how hierarchical edges can be used to capture the topology in a triangle mesh with faces, triangle edges and vertices. The left hand side shows two triangles that share the edge e_3. Each triangle has three vertices that are connected in pairs by graph edges $\{e_1, e_2, e_3, e_4, e_5\}$ that represent triangle edges. However, these edges are themselves connected by face edges $\{f_1, f_2\}$, thus forming the hierarchy shown on the right hand side. By representing both triangle edges and faces we can store different quantities on them. Moreover, it becomes possible to accelerate typical mesh queries such as finding the adjacent faces of a face by inserting topological indices, such as a half-edge index, into the graph structure.

4.2 Tensors with Blocks

We use zero indices to index a scalar, one to index a vector and two to index a matrix. Tensors generalize scalars, vectors and matrices to an arbitrary number of indices. We call the number of indices required to index into a tensor its **order**. Thus, scalars are 0th-order tensors, vectors are 1st-order tensors, and matrices are 2nd-order tensors. Further, we refer to the nth tensor index as its nth dimension. Thus, the first dimension of an $m \times n$ matrix is the rows m, while the second dimension is the columns n.

The dimensions of a Simit tensor are sets: either integer ranges or hypergraph sets. Thus, a Simit vector is more like a dictionary than an array, and an n-order tensor is an n-dimensional dictionary, where an n-tuple of hypergraph set elements map to a tensor component. For example, Figure 6 (center) depicts a matrix B whose dimensions are (V, U), where $V = \{\mathrm{v}_1, \mathrm{v}_2\}$ and $U = \{\mathrm{u}_1, \mathrm{u}_2, \mathrm{u}_3\}$. We can index into the matrix using an element from each set. For example, $B(\mathrm{v}_2, \mathrm{u}_2)$ locates the gray component.

Simit tensors can also be blocked. In a blocked tensor each dimension consists of a hierarchy of sets. For example, a hypergraph set could map hyperedges to tensor blocks, where each block is described by an integer range. Blocked tensors are indexed using **hierarchical indexing**. This means that if we index into a blocked tensor using an element from the top set of a dimension, the result is a tensor block. Figure 6 (right) shows a blocked matrix C whose dimensions are $(V \times 3, U \times 4)$, which means there are $|V| \times |U|$ blocks of size 3×4. As before, we can index into the matrix using an element from each set, $C(\mathrm{v}_2, \mathrm{u}_2)$, but now the index operation

results in the 3×4 grey block matrix. If we index into the matrix block, $C(\mathrm{v}_2, \mathrm{u}_2)(2, 2)$ we locate the dark grey component. In addition to being convenient for the programmer, blocked tensors let Simit produce efficient code. By knowing that a sparse matrix consists of dense inner blocks, Simit can emit dense inner loops for sparse matrix-vector multiplies with that matrix.

4.3 Tensor Assembly using Maps

A **tensor assembly** is a map from the triple (S, f, r) to one or more tensors, where S is a hypergraph set, f an **assembly function**, and r an associative and commutative reduction operator. The tensor assembly applies the assembly function to every element in the hypergraph set, producing per-element tensor contributions. The tensor assembly then aggregates these tensor contributions into a global tensor, using the reduction operator to combine values. The result of the tensor assembly is one or more global tensors, whose dimensions can be the set S or any of its endpoints. The diagram in Figure 7 shows this process. On the left is a graph where the edges $E = \{e_1, e_2\}$ connect the vertices $V = \{v_1, v_2, v_3\}$. The function `f` is applied to every edge to compute contributions to the global $V \times V$ matrix. The contributions are shown in grey and the tensor assembly aggregates them by adding the per-edge contribution matrices.

Assembly functions are pure functions whose arguments are an element and its endpoints, and that return one or more tensors that contain the element's global tensor contributions. The arguments of an assembly function are supplied by a tensor assembly as it applies the function to every element of a hypergraph set, and the same tensor assembly combines the contributions of every assembly function application. The center of Figure 7 shows code for `f`: a typical assembly function that computes the global matrix contributions of a 2-edge and its vertex endpoints. The function takes as arguments an edge `e` of type `Edge`, and a tuple `v` that contains `e`'s two `Vertex` endpoints. The result is a $V \times V$ matrix with 3×3 blocks as shown in the figure. Notice that `f` can only write to four location in the resulting $V \times V$ matrix, since it has access to only two vertices. In general, an assembly function that maps a c-edge to an n-dimensional tensor, can write to exactly c^n locations in the tensor. We call this property **coordinate-free indexing**, since each assembly function locally computes and writes its matrix contributions to the global matrix using opaque indices (the vertices) without regards to where in the matrix those those contributions end up. Further, since the global matrix is blocked, the 3×3 matrix `k` can be stored into it with one assignment, by only specifying block coordinates and not intra-block coordinates. As described in Section 4.2 we call this property hierarchical indexing, and the resulting *coordinate-free hierarchical indexing* removes a large class of indexing bugs, and makes assembly functions easy to write and read.

We have so far discussed the functional semantics of tensor assemblies, but it is also important to consider their performance semantics. The way they are defined above, if executed literally, would result in very inefficient code where ultra-sparse tensors are created for every edge, followed by a series of tensor additions. However, as discussed in Section 6, the tensor assembly abstraction lets Simit's compiler produce code that stores tensor blocks on the graph elements corresponding to one of the tensor dimensions. Thus, memory can be pre-allocated and indexing structures pre-built, and assembly becomes as cheap as computing and storing blocks in a contiguous array. As discussed in Section 9, the tensor assembly construct lets the Simit compiler know where global vectors and matrices come from, which lets it emit fast in-place code.

4.4 Tensor Computation using Index Expressions

So far, we have used linear algebra to compute with scalars, vectors and matrices. Linear algebra is familiar and intuitive for programmers, so we provide it in the Simit language, but it has two important drawbacks. First, it does not extend to higher-order tensors. Second, it is riddled with operators that have different meanings depending on the operands, and does not let us cleanly express computations that perform multiple operations simultaneously. This makes linear algebra ill-suited as compute operators in the Simit programming model. Instead we use index expressions, which are a generalization of tensor index notation [Ricci-Curbastro and Levi-Civita 1901] in expression form. Index expressions have all of the properties we seek, and as an added benefit we can build all the basic linear algebra operations on top of them. Thus, programmers can work with familiar linear algebra when that is convenient, and the linear algebra can be lowered to index expressions that are easier to optimize and generate efficient code from (see Section 7).

An index expression computes a tensor and consists of a scalar expression and one or more **index variables**. Index variables are used to index into tensor operands. There are two types: **free variables** and **reduction variables**. Free variables determine the dimensions of the tensor resulting from the index expression, while reduction variables combine values. Thus, an index expression has the form:

```
(free-variable*) reduction-variable* scalar-expression
```

where `scalar-expression` is a normal scalar expression, whose operands that are typically indexed tensors.

Free index variables are variables that can take the values of a tensor dimension: an integer range, a hypergraph set, or a hierarchical set as shown in Figure 6. The values an index variable can take are called its range. The range of the free index variables of an index expression determine the dimensions of the resulting tensor. To compute the value of one component of this tensor the index variables are bound to the component's coordinate, and the index expression is evaluated. To compute every component of the resulting tensor, the index expression is evaluated for every value of the set product of the free variables' ranges. For example, consider an index expression that computes a vector addition:

```
(i) a(i) + b(i)
```

In this expression `i` is a free index variable whose range is implicitly determined by the dimensions of the vectors `a` and `b`. Note that an index variable can only be used to index into a tensor dimension that is the same as its range. Thus, the vector addition requires that the dimensions of `a` and `b` are the same. Further, the result of this index expression is also a vector whose dimension is the range of `i`. Next, consider a matrix transpose:

```
(i,j) A(j,i)
```

Here `i` and `j` are free index variables whose ranges are determined by the second and first dimensions of `A` respectively. As expected, the dimensions of the resulting matrix are the reverse of `A`'s, since the order of the free index variables in the list determines the order of the result dimensions. Finally, consider an index expression that adds `A` and $\mathtt{B}^T$:

```
(i,j) A(i,j) + B(j,i)
```

Since index variables ranges take on the values of the dimensions they index, this expression requires that the first and second dimensions of `A` are the same as the second and first dimensions of `B` respectively. This example shows how one index expression can simultaneously evaluate multiple linear algebra operations.

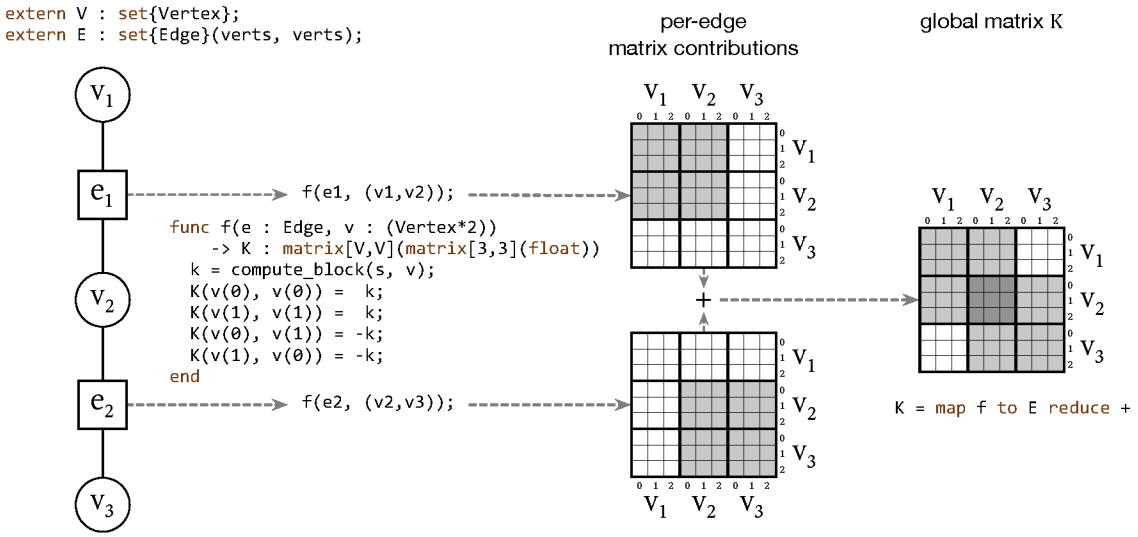


Fig. 7: Global matrix `K` assembly. The assembly map (on the right) applies `f` to every edge in `E`, and sums the resulting matrix contributions. `f` computes a block `k` that is stored into the positions of `K` that correspond to the current edge's endpoints. Since each edge only has two endpoints, `f` can only store into four locations of the matrix. As a result, the top right and lower left entries of the final matrix `K` are empty, since no edge connects v_1 and v_3. Finally, note the collision at (v_2, v_2) since v_2 is an endpoint of both e_1 and e_2.

Reduction variables, like free variables, range over the values of tensor dimensions. However, unlike free variables, reduction variables do not contribute to the dimensions of the resulting tensor. Instead, they describe how a range of computed values must be *combined* to produce a result component. How these values are combined is determined by the reduction variables's reduction operator, which must be an associative and commutative operation. For example, a vector dot product $\sum_i a(i) * b(i)$ can be expressed using an addition reduction variable:

```
+i a(i) * b(i)
```

As with free index variables, the range of `i` is implicitly determined by the dimension of `a` and `b`. However, instead of resulting in a vector with one component per value of `i`, the values computed for each `i` are added together, resulting in a scalar. Free index variables and reduction index variables can also be combined in an index expression, such as in the following matrix-matrix multiplication:

```
(i, k) +j A(i,j) * B(j,k)
```

The two free index variables `i` and `k` determine the dimensions of the resulting matrix.

In this section we showed how index expressions can be used to express linear algebra in a simpler and cleaner framework. We showed a simple example where two linear algebra operations (addition and transpose) were folded into a single index expression. As we will see in Section 7, index expressions make it easy for a compiler to combine basic linear algebra expressions into arbitrarily complex expressions that share index variables. After code generation, this results in fewer loop nests and less memory traffic.

5. INTERFACING WITH SIMIT

To use Simit in an application there are four steps:

(1) Specify the structure of a system as a hypergraph with vertex sets and edge sets, using the C++ Set API described in Section 5.1,

(2) Write a program in the Simit language to compute on the hypergraph, as described in Sections 3 and 4,

(3) Load the program, bind hypergraph sets to it, and compile it to one or more `Function` objects, as described in Section 5.2,

(4) Call a `Function` object's `run` method once for each required program execution (e.g., a time step, static solve, etc.), as described in Section 5.2.

Collision detection, fracturing, and topology changes cannot currently be expressed in the Simit language. However, they can be computed using C++ and incorporated into the simulation using Simit's Set API to dynamically add or remove vertices and edges in the hypergraph between each Simit program execution. For example, external collision detection code between time steps can express detected contacts as edges between colliding elements.

5.1 Set API

Simit's Set API consists of of C++ classes and functions that create hypergraph sets with tensor fields. The central class is the `Set` class, which creates sets with any number of endpoints, that is, both vertex sets and edge sets. When a `Set` is constructed, the `Set`'s endpoints are passed to the constructor. Next, fields can be added using the `Set::addField` method and elements using the `Set::add` method.

The following code shows how to use the Simit Set API to construct a pyramid from two tetrahedra that share a face. The vertices and tetrahedra are assigned fields that match those in Section 3:

```
Set verts;
Set tets(verts, verts, verts, verts);

// create fields (see the FEM example in Figure 2)
FieldRef<double,3> x  = verts.addField<double,3>("x");
FieldRef<double,3> v  = verts.addField<double,3>("v");
FieldRef<double,3> fe = verts.addField<double,3>("fe");

FieldRef<double>     u = tets.addField<double>("u");
FieldRef<double>     l = tets.addField<double>("l");
FieldRef<double>     W = tets.addField<double>("W");
FieldRef<double,3,3> B = tets.addField<double,3,3>("B");
```

```
// create a pyramid from two tetrahedra
Array<ElementRef> v = verts.add(5);
ElementRef t0 = tets.add(v(0), v(1), v(2), v(4));
ElementRef t1 = tets.add(v(1), v(2), v(3), v(4));

// initialize fields
x(v0) = {0.0, 1.0, 0.0};
// ...
```

First, we create the `verts` vertex set and `tets` edge set, whose tetrahedron edges each connects four `verts` vertices. We then add to the `verts` and `tets` sets the fields from the running example in Section 3. The `Set::addField` method is a variadic template method whose template parameters describe the tensors stored at each set element. The first template parameter is the tensor field's component type (`double`, `int`, or `boolean`), followed by one integer literal per tensor dimension. The integers describe the size of each tensor dimension; since the `x` field above is a position vector there is only one integer. To add a 3×4 matrix field we would write: `addField<double,3,4>`. Finally, we create five vertices and the two tetrahedra that connect them together, and initialize the fields.

5.2 Program API

Once a hypergraph has been built (Section 5.1) and a Simit program written (Sections 3 and 4), the Program API can be used to compile and run the program on the hypergraph. To do this, the programmer creates a `Program` object, loads Simit source code into it, and compiles an exported `func` in the source code to a `Function` object. Next, the programmer binds hypergraph sets to externs in the `Function` objects' Simit program, and the `Function::run` method is called to execute the program on the bound sets.

The following code shows how to load the FEM code in Figure 2, and run it on tetrahedra we created in Section 5.1:

```
Program program;
program.loadFile("fem_statics.sim");

Function func = program.compile("main");
func.bind("verts", &verts);
func.bind("tets",  &tets);

func.runSafe();
```

In this example we use the `Function::runSafe` method, which lazily initializes the function. For more performance the initialization and running of a function can be split into a call to `Function::init` followed by one or more calls to `Function::run`.

6. RUNTIME DATA LAYOUT AND EXECUTION

In Sections 3 and 4 we described the language and abstract data structures (hypergraphs and tensors) that a Simit programmer works with. Since the abstract data structures are only manipulated through global operations (tensor assemblies and index expressions) the Simit system is freed from implementing them literally, an important property called physical data separation [Codd 1970]. Simit exploits this separation to compile global operations to efficient local operations on compact physical data structures that look very different from the graphs and tensors the programmer works with. In the rest of this section we go into detail on how the current Simit implementation lays out data in memory and what kind of code it emits. This is intended to give interested readers a sense of how the physical data separation lets Simit pin global vectors and matrices to graph vertices and edges for in-place computation. It also demonstrates how the tensor assembly construct lets Simit use the graph as the index for matrices, and how the blocked matrix types let Simit emit dense inner loops when computing with sparse matrices.

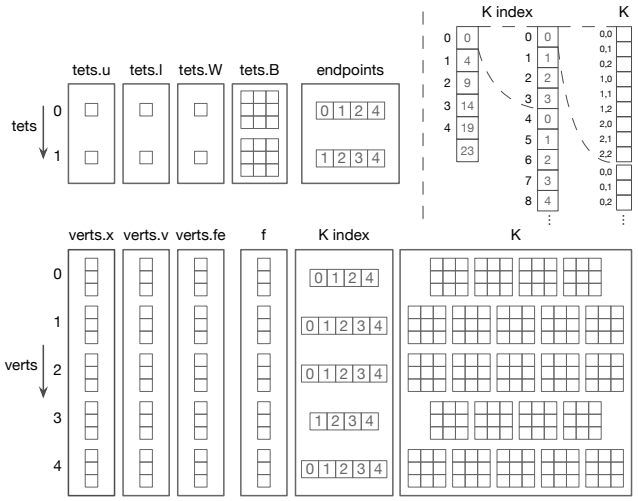


Fig. 8: All the data stored for the two `tets` tetrahedra constructed in Section 5.1. The global vector `f` and matrix `K` from Figure 2 are stored on the `verts` set, since `verts` is their first dimension. The table is stored by columns (struct of arrays) and matrix blocks are stored row major. `K` is sparse so it is stored as a segmented array (top right), consisting of an array of row vertices, an array of column indices, and a value array. `K` is assembled from the `tets` set so its sparsity is known. The K index is therefore precomputed and is shared with other matrices assembled from the same edge set.

6.1 Storage

As described in Section 4, Simit graphs consist of vertex sets and edge sets, which are the same except that edges have endpoints. A Simit set stores its size (one integer) and field data. In addition, edge sets store a pointer to each of its n endpoint sets and for each edge, n integer indices to the edge's endpoints within those endpoint sets. Figure 8 (top) shows all the data stored on each `Tet` in the `tets` set we built in Section 5.1 for the FEM example in Figure 2. Each `Tet` stores the fields `u`, `l`, `W`, and `B`, as well as an `endpoints` array of integer indexes into the `verts` set. The set elements and their fields form a table that can be stored by rows (arrays of structs) or by columns (structs of arrays). The current Simit implementation stores this table by columns, so each field is stored separately as a contiguous array with one scalar, vector, dense matrix or dense tensor per element. Furthermore, dense matrices and tensors are stored in row-major order within the field arrays.

Global vectors and matrices are also stored as fields of sets. Specifically, a global vector is stored as a field of its dimension set, while a global matrix is stored as a field of one of its dimension sets. That is, either matrix rows are stored as a field of the first matrix dimension or matrix columns are stored as a field of the second matrix dimension. This shows the equivalence in Simit of a set field and global vector whose dimension is a set— a key organizing property. Figure 8 (bottom) shows all the data stored on each `Vertex` in the `verts` set, including the global vector `f` and global matrix `K` from the `newton_method` function in Figure 2. Since `K` is sparse, its rows can have different sizes and each row is therefore stored as an array of column indices (`K index`) and a corresponding array of data (`K`). The column indices of `K` are the `verts` vertices that each row vertex can reach through non-empty matrix components. Since the `K` matrix was constructed from a tensor assembly over the `tets` edge set, the neighbors through the matrix are the same as the neighbors through `tets` and can be precomputed. Since global matrix indices and values are set fields with a different number of

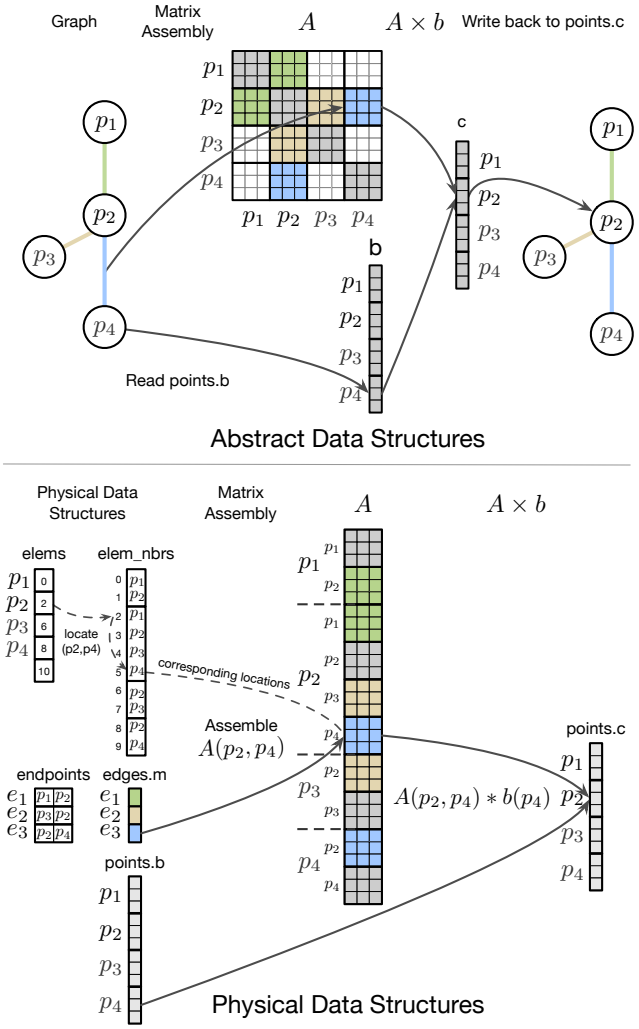


Fig. 9: Top: a tensor assembly assembles the abstract matrix A. The field `points.b` is read from the points set and multiplied by A, into c. Finally, c is written to field `points.c`. Bottom: the neighbor index structure `rowstart` and `neighbors` arrays are used to store A's values to a segmented array. The array is then multiplied in-place by `points.b` to compute `points.c`.

entries per set element, they are stored in a segmented array, as shown for `K` in Figure 8 (upper right). Thus, Simit matrix storage is equivalent to Blelloch's segmented vectors [Blelloch 1990] and the BCSR (Blocked Compressed Sparse Row) matrix storage format.

6.2 Execution

A Simit tensor assembly map statement is compiled into a loop that computes the tensor values and stores them in the global tensor data structures. The loop iterates over the map's target set and each loop iteration computes the local contributions of one target set element using the map function, which is inlined for efficiency. Equivalent sequential C code to the machine code generated for the map statement on line 50 of Figure 2 is:

```
for (int t=0; t<tets.len; t++) {
  for (int i=0; i<4; i++) {
    double[3] tmp = compute_tet_force(t,v,i); // inlined
    for (int j=0; j<3; j++) {
      f[tets.endpoints[t*4 + i]*3 + j] += tmp[j];
    }
  }
}
```

The outer loop comes from the map statement itself and iterates over the tetrahedra. Its loop body is the inlined `tet_force` function, which iterates over the four endpoints of the tetrahedra and for each endpoint computes a tet force that is stored in the `f` vector. A global matrix is assembled similarly with the exception that the location of a matrix component must be computed from the matrix's index array as follows (taken from the Simit runtime library):

```
int loc(int v0, int v1, int *elems, int *elem_nbrs) {
  int l = elems[v0];
  while(elem_nbrs[l] != v1) l++;
  return l;
}
```

The `loc` function turns a two-dimensional coordinate into a one-dimensional array location, given a matrix index consisting of the arrays `elems` and `elem_nbrs`. It does this by looking up the location in `elem_nbrs` where the row (or column) `v0` starts. That is, it finds the correct segment of `v0` in the segmented array `elem_nbrs`. It then scans down this segment to find the location of the element neighbor `v1`, which is then returned.

Figure 9 shows an end-to-end example where a matrix is assembled from a normal graph and multiplied by a vector field of the same graph. The top part shows the abstract data structure views that the programmer works with, which were described in Section 4. The arrows shows how data from the blue edge is put into the matrix on matrix assembly, how data from the p_4 vertex becomes the p_4 block of the `b` vector, and how the block in the $(\mathrm{p}_2, \mathrm{p}_4)$ matrix component is multiplied with the block in the p_4 `b` vector component to form the p_2 `c` vector component when the `A` matrix is multiplied with `b`. The bottom part shows the physical data structures; the vertex set has a field `points.b` and the edge set has a field `edges.m`. The stippled arrows show how the `loc` function is used to find the correct location in the array of `A` values when storing matrix contributions of the blue edge. The full arrows show how the `edges.m` field is used to fill in values in the $(\mathrm{p}_2, \mathrm{p}_4)$ matrix component (the sixth block of the `A` array), and how this block is multiplied directly with the p_4 `points.b` vector component to form the p_2 `points.c` vector component when `A` is multiplied with `b`.

Index expressions are compiled to loop nests. For the matrix-vector multiplication in Figure 9 the following code is emitted:

```
for (int i=0; i<points.len; i++) {
  for (int ij=elems[i]; ij<elems[i+1]; ij++) {
    int j = elems_nbrs[ij];
    for (int ii=0; ii<3; ii++) {
      int tmp = 0;
      for (int jj=0; jj<3; jj++) {
        tmp += A[ij*9 + ii*3 + jj] * points.b[j*3 + jj];
      }
      points.c[i*3 + ii] += tmp;
    }
  }
}
```

This code is equivalent to the standard BCSR matrix-vector multiplication. Two sparse outer loop iterate over `A`'s index structure and the two inner loops iterate over the blocks. In the innermost block a (3×3) block of `A` is retrieved using the `ij` variable, which corresponds to a matrix location.

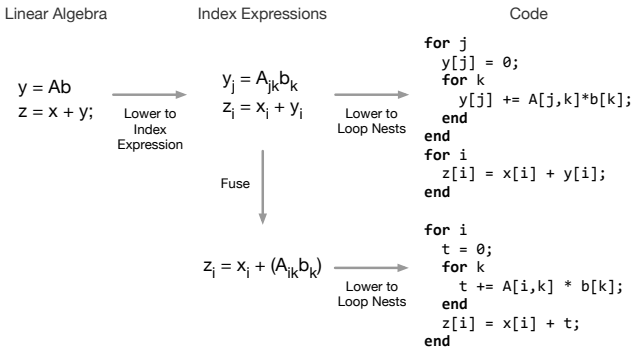


Fig. 10: Code generation from index expressions. First, linear algebra operations are parsed into index expressions (left). These can be directly converted to loops (top) or they can be fused (bottom). Fusing index expressions results in fewer loop nests and fewer temporary variables (the `y` vector is replaced by the scalar temporary `t`).

6.3 Graph Changes

As discussed in Section 1, certain features require the graph topology to change between time steps, for example to implement collisions, cuts or remeshing. When graphs change, neighbor indices necessarily have to be updated. This is true for any system, but a system like Simit permits indices to only be updated when necessary and as much as necessary, and it also allows several assembled matrices to share the same index. With the current Simit implementation indices are recomputed if the graph topology changes, but future work includes incrementally updating indices.

7. COMPILER OVERVIEW

The Simit compiler is implemented as a C++ library. Below we list stages that Simit code goes through before it is emitted as binary code using LLVM. During parsing, an internal compiler representation containing maps, index expressions and control flow constructs is built. As discussed in Section 4.4, index expressions are Simit's way to represent computation on tensors and is similar to tensor index notation [Ricci-Curbastro and Levi-Civita 1901]. Linear algebra expressions are turned into index expressions during parsing. Figure 10 shows a linear algebra expression that adds the vectors x and $y = Ab$. This linear algebra is first lowered to two index expressions, the first of which is $y_j = A_{jk}b_k$, where j is a free variable and k is a reduction variable that sums the product of A_{jk} and b_k for each k. This example uses the Einstein convention [Einstein 1916], where variables repeated within a term are implicitly sum reductions.

A Simit program goes through four major transformation phases before it is turned into machine code using the LLVM compiler infrastructure. Parallel GPU code generation requires one additional phase. The idea behind these phases is to lower high-level constructs like assembly maps, index expressions and multidimensional tensor accesses to simple loops with 1D array accesses that are easy to emit as low level code.

Index Expression Fusing. Index expressions are fused when possible to combine operations that would otherwise require tensor intermediates. Figure 10 shows an example of this phase in action; the lower portion of the figure shows how fused index expressions lead to fused loops. Without the use of index expressions, this kind of optimization requires using heuristics to determine when and how linear algebra operations can be combined. Simit can easily eliminate tensor intermediates and perform optimizations that are difficult to do in the traditional linear algebra library approach, even with the use of expression templates.

Map Lowering. Next, the compiler lowers map statements by inlining the mapped functions into loops over the target set of the map. In this process, the Simit compiler uses the reduction operator specified by the map to combine sub-blocks from different elements. Thus, the map is turned into inlined statements that build a sparse system matrix (shown in Figure 9) from local operations.

Index Expression Lowering. In this phase, all index expressions are transformed into loops. For every index expression, the compiler replaces each index variable with a corresponding dense or sparse loop. The index expression is then inserted at the appropriate places in the loop nest, with index variables replaced by loop variables. This process is demonstrated in the middle and right panes of Figure 10.

Lowering Tensor Accesses. In the next phase, tensor accesses are lowered. The compiler takes statements that refer to multidimensional tensor locations and turns them into concrete array loads and stores. A major optimization in this phase is to use context information about the surrounding loops to make sparse matrix indexing more efficient. For example, if the surrounding loop is over the same sets as the sparse system matrix, we can use existing generated variables to index into the sparse matrix instead of needing to iterate through the column index of the neighbor data structure for each read or write.

Code Generation. After the transformation phases, a Simit program consists of imperative code, with explicit loops, allocations and function calls that are easy to turn into low-level code. The code generation phase turns each Simit construct into the corresponding LLVM operations, using information about sets and indices to assist in generating efficient code. Currently, the backend calls LLVM optimization passes to perform inter-procedural optimization on the Simit program, as well as other standard compiler optimizations. Only scalar code is generated; we have not yet implemented vectorization or parallelism, and LLVM's auto-vectorization passes cannot automatically transform our scalar code into vector code. Future work will implement these optimizations during code generation, prior to passing the generated code to LLVM's optimization passes. Our index expression representation is a natural form in which to perform these transformations.

GPU Code Generation. Code generation for GPU targets is performed as an alternative code generation step specified by the user. Making use of Nvidia's NVVM framework lets us generate code from a very similar LLVM structure as the CPU-targeted code generation. Because CUDA kernels are inherently parallel, a GPU-specific lowering pass is performed to translate loops over global sets into parallel kernel structures. Broadly, to convert global for-loops into parallel structures, reduction operations are turned into atomic operations and the loop variable is replaced with the CUDA thread ID. Following this, we perform a GPU-specific analysis to fuse these parallel loops wherever possible to reduce kernel launch overhead and increase parallelism. Using a very similar pipeline for CPU and GPU code generation helps us ensure that behavior on the GPU and CPU is identical for the code structures we generate.

8. RESULTS

To evaluate Simit, we implemented three realistic simulation applications, Implicit Springs, Neo-Hookean FEM and Elastic Shells, using Simit, Matlab and Eigen. In addition, we implemented Neo-Hookean FEM using SOFA and Vega, two hand-optimized state-of-the-art real-time physics engines, and we also implemented Implicit Springs using SOFA. Note that Vega did not support Implicit Springs, and neither Vega nor SOFA supported Elastic Shells.

We then conducted five experiments that show that:

(1) Using the traditional approaches we evaluate you get better performance by writing more code. With Simit you can get both performance and productivity. (Section 8.3)

(2) Simit programs can be compiled to GPUs with no change to the source code to get 4-20× more performance. (Section 8.4)

(3) Simit programs that use solvers are memory bound. (Section 8.5)

(4) Simit programs scale well with increased dataset sizes. (Section 8.6)

(5) A CG Solver written in Simit is competitive with many optimized iterative solvers written in C++. (Section 8.7)

We also implemented two additional FEM variants: a Hexahedral Corotational FEM and a Tetrahedral Laplace FEM. We used the former to compare against the hand-optimized Hexahedral Corotational FEM implementation of Dick et al. [2011] (Section 8.8), and the latter to compare against one of the tutorial applications of the FreeFem++ PDE FEM Language (Section 8.9).

All CPU timings are taken on an Intel Xeon E5-2695 v2 at 2.40GHz with 128 GB of memory running Linux. All CPU measurements except AMGCL are single-threaded—none of the other libraries we compare to support multi-threaded CPU execution, nor does the current Simit compiler, though parallelization will be added in the future. The Simit and cuSPARSE GPU timings in Section 8.4 are taken on an Nvidia Titan GK210, while the Simit and Dick et al. GPU timings in Section 8.8 are taken on an Nvidia Geforce Titan Z.

8.1 Applications

We implemented three simulation applications with different edge topologies and computational structure, and paired each with a suitable data set (bunny, dragon and cloth).

For Implicit Springs and Neo-Hookean FEM we chose to implement the CG solver in Simit instead of using an external solver library. The reason for this is that Simit offers automatic portability to GPUs, natively compiles SpMV operations to be blocked with a dense inner loop (see Section 9), and because it avoids data translation going from Simit to the external solver library. To evaluate the performance benefit of the Simit CG, we ran an experiment where Simit instead used Eigen to perform the CG solve. This resulted in a 30% slowdown, due to data translation and because Eigen does not take advantage of block structure (see Section 8.7 for details).

8.1.1 *Implicit Springs.* Our first example is a volumetric elasticity simulation using implicit springs. We tetrahedralized the Stanford bunny to produce 37K vertices and 220K springs, and passed this to Simit as a vertex set connected by an edge set. Our implementation uses two assembly maps—one on the vertex set to compute the mass and damping matrices, and one on the edge set to compute the stiffness matrix. To solve for new vertex velocities, we implement a linearly-implicit time stepper and use the method of conjugate gradients (CG) as our linear solver. Per common practice, when implementing a performance-sensitive implicit solver, we limit the maximum number of conjugate gradients iterations (we choose 50).

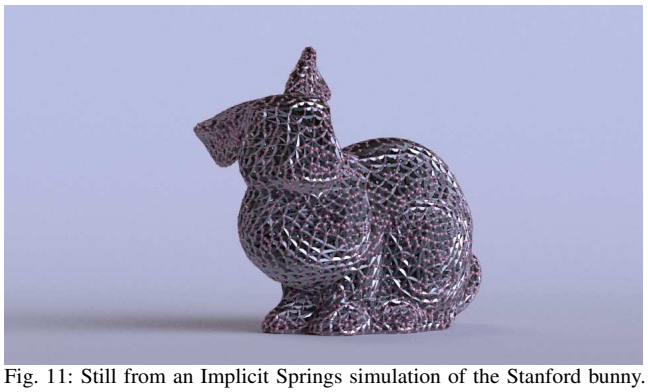


Fig. 11: Still from an Implicit Springs simulation of the Stanford bunny. The bunny consists of 36,976 vertices and 220,147 springs, which can be simulated by Simit on a GPU at 12 frames per second. Only surface vertices and springs are shown. The Simit code is only 93 lines, which includes a conjugate gradient solver implementation.

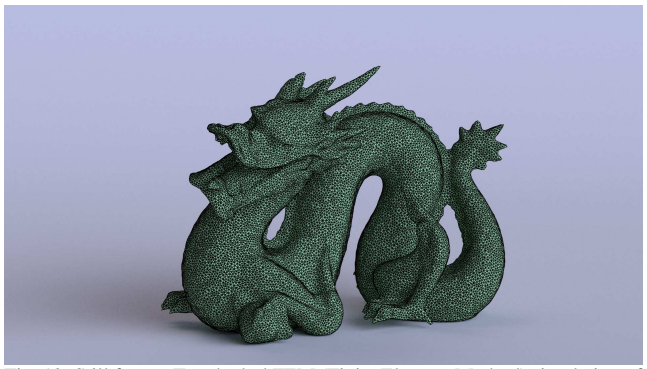


Fig. 12: Still from a Tetrahedral FEM (Finite Element Method) simulation of a dragon with 46,779 vertices and 160,743 elements, using the Neo-Hookean material model. Simit performs the simulation at 11 frames per second with only 154 non-comment lines of code shown in Appendix A. This includes 15 lines for global definitions, 14 lines for utility functions, 69 lines for local operations, 23 lines to implement CG and 13 lines for the global linearly-implicit timestepper procedure to implement the simulation, as well as 20 lines to precompute tet shape functions.

8.1.2 *Tetrahedral Neo-Hookean FEM.* Our second example is one of the most common methods for animating deformable objects, tetrahedral finite elements with linear shape functions. We chose tetrahedral meshes here because they are easy to conform to geometry, which makes accurate treatment of boundary conditions simpler, but we also evaluate hexahedral meshes in Section 8.1.4. We use the non-linear Neo-Hookean material model [Mooney 1940], as it is one of the standard models for animation and engineering, and we set the stiffness and density of the model to realistic physical values. The Simit implementation uses three maps, one to compute forces on each element, one to build the stiffness matrix and one to assemble the mass matrix, and then solves for velocities. We then use the conjugate gradient method to solve for the equations of motion, again relying on an implementation in pure Simit.

8.1.3 *Elastic Shells.* For our third example, we implemented an elastic shell code and used it to simulate the classic example of a rectangular sheet of cloth draping over a rigid, immobile sphere.

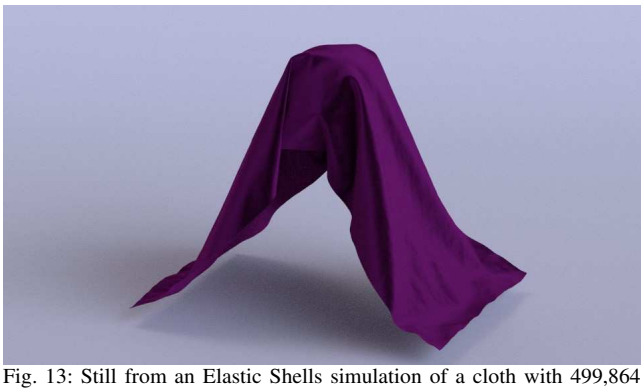


Fig. 13: Still from an Elastic Shells simulation of a cloth with 499,864 vertices, 997,012 triangle faces and 1,495,518 hinges. Elastic shells require two hyperedge sets: one for the triangle faces and one for the hinges. The Simit implementation ran at 15 frames per second on a GPU.

The input geometry is a triangle mesh with 997,012 faces connecting 499,864 vertices, and is encoded as a hypergraph using one vertex set (the mesh vertices) and two hyperedge sets: one for the triangle faces, which are the stencil for a constant-strain Saint Venant-Kirchhoff stretching force; and one for pairs of triangles meeting at a *hinge*, the stencil for the Discrete Shells bending force of Grinspun et al [Grinspun et al. 2003]. The bending force is a good example of the benefit of specifying force computation as a function mapped over an edge set: finding the two neighboring triangles and their vertices given their common edge, in the correct orientation, typically involves an intricate and bug-prone series of mesh traversals. Simit provides the bending kernel with the correct local stencil automatically. The Simit implementation uses a total of five map operations over the three sets to calculate forces and the mass matrix before updating velocities and positions for each vertex using explicit Velocity Verlet integration.

8.1.4 *Hexahedral Corotational FEM.* We implemented a Hexahedral Corotational finite element method to simulate deformable solids with arbitrary non-linear hyperelastic materials. Our implementation follows the implementation of Dick et al. [2011] as closely as possible. We precompute and store a 24×24 standard stiffness matrix for an input mesh given its element size and Poisson's ratio. During simulation, we compute a 3×3 rotation matrix for each element. The main difference between our implementation and theirs is the linear solver; we use a CG solver instead of multigrid, since we do not yet support multigrid in Simit. The Simit implementation required 133 lines of Simit code, plus 24 lines for the CG solver.

8.1.5 *Tetrahedral Laplace FEM.* We also investigated the performance of Simit on less complicated partial differential equations. Specifically, we implemented a tetrahedral finite element method for solving the Laplace equation, requiring only a single map to compute the system matrix. The resulting discretized linear system is solved using our native Simit CG solver. We use this to examine the steady state heat distribution inside of arbitrary 3D domains in the presence of Dirichlet boundary conditions.

8.2 Languages and Libraries

We implemented each application in Matlab and in C++ using the Eigen high-performance linear algebra library. In addition, we used the SOFA simulation framework to implement Implicit Springs and Neo-Hookean FEM, and the Vega FEM simulator library to implement Neo-Hookean FEM.

8.2.1 *Matlab.* Matlab is a high-level language that was developed to make it easy to program with vectors and matrices [MATLAB 2014]. Matlab can be seen as a scripting language on top of high-performance linear algebra operations implemented in C and Fortran. Even though Matlab's linear algebra operations are individually very fast, they don't typically compose into fast simulations. The main reasons for this is Matlab's high interpretation overhead, and the fact that individually optimized linear algebra foregoes opportunities for fusing operations (see Sections 7 and 9).

8.2.2 *Eigen.* Eigen is an optimized and vectorized linear algebra library written in C++ [Guennebaud et al. 2010]. To get high performance it is uses template meta-programming to produce specialized and vectorized code for common operations, such as 3×3 matrix-vector multiply. Furthermore, Eigen defers execution through its object system, so that it can fuse certain linear algebra operations such as element-wise addition of dense vectors and matrices.

8.2.3 *SOFA.* SOFA is an open source framework, originally designed for interactive, biomechanical simulation of soft tissue [Faure et al. 2007]. SOFA's design is optimized for use with iterative solvers. It uses a scene graph to model the interacting objects in a simulation and, during each solver iteration, visits each one in turn, aggregating information such as applied forces. Using this traversal SOFA avoids forming global sparse matrices which is the key to its performance.

8.2.4 *Vega.* Vega is a graphics-centric, high-performance finite element simulation package [Sin et al. 2013]. Vega eschews special scene management structures in favor of a general architecture that can be used with iterative or direct solvers. It achieves high performance using optimized data-structures and clever rearrangement of material model computations to reduce operation count.

8.3 Simit Productivity and Performance

The general trend in the traditional systems we evaluate is that you get more performance by writing more code. Table I shows this for the three applications from Section 8.1. For each application and each language/library we report the performance (milliseconds per frame, where each frame is one time step), source lines and memory consumption. For two applications we vectorized the Matlab code to remove all loops (Matlab Vec). This made the Matlab code about one order of magnitude faster, but took 9 and 16 hours of additional development time for domain experts who are proficient with Matlab, and made the code very hard to read (see supplemental material).

For example, the Eigen Implicit Springs implementation is more than twice the code of the Matlab implementation, but runs 15 times faster. In our experiments higher performance meant more code had to be written in a lower-level language. Simit, however, breaks this tradeoff and gives high performance with few lines of code. With one exception, the Simit implementation is faster than the fastest alternative we found, while being fewer lines of code than Matlab. For example, Simit performs better than Vega, an optimized FEM library (Section 8.2.4), with $7\times$ fewer lines of code, and compiles to GPUs for 4-20$\times$ more performance (Section 8.4). Furthermore, we plan to vectorize and multi-thread Simit in the future, which will further increase its performance.

For Elastic Shells, Simit is 19% slower than Eigen. This is due to the application's explicit method to simulate cloth, which benefits from Eigen's vectorized dense vector routines. Simit does not yet support vectorization. However, by using a GPU Simit can drastically increase performance on this code (see Section 8.4) with no change to the source code which is half the size of the Eigen CPU implementation.

Table I. : Comparison of three applications implemented with Matlab, Vectorized Matlab, Eigen, hand-optimized C++ (SOFA and Vega) and Simit, showing the productivity and performance of Simit. For example, the Simit Neo-Hookean FEM is just 154 lines of code (includes CG solver), but simulates a frame with 160,743 tetrahedral elements in just 0.9 seconds using a single non-vectorized CPU thread. For each implementation we report non-comment source lines of code, milliseconds per frame and peak memory in megabytes (1024^2 bytes), as well as the size of each number relative to Simit. The trend is that users get better performance by writing more code, however, Simit provides both good performance and productivity. For example, the Simit Implicit Springs is the shortest implementation at 93 lines of code, yet runs fastest at 0.48 seconds per frame. Matlab ran out of memory running the cloth simulation. The applications were run on the bunny, dragon and cloth data sets respectively with double precision floating point on an Intel Xeon E5-2695 v2 running at 2.40GHz with 128 GB of memory. Applications that use CG were run with 50 CG iterations per frame.

		ms per frame		Source lines		Memory (MB)	
Implicit Springs	Matlab	13,576	28.2×	142	1.5×	1,059	6.5×
	Matlab Vec	2,155	4.5×	230	2.5×	1,297	8.0×
	Eigen	898	1.9×	314	3.4×	339	2.1×
	SOFA	490	1.0×	1,404	15.1×	94	0.6×
	Simit	481	1.0×	**93**	1.0×	163	1.0×
Neo-Hookean FEM	Matlab	213,797	237.2×	234	1.5×	1,564	10.3×
	Matlab Vec	16,080	17.8×	293	1.9×	53,949	354.9×
	Eigen	2,268	2.5×	363	2.3×	626	4.1×
	SOFA	1,315	1.5×	1,541	10.0×	324	2.1×
	Vega	1,025	1.1×	1,080	7.0×	614	4.0×
	Simit	901	1.0×	**154**	1.0×	152	1.0×
Elastic Shells	Matlab			203	1.1×	OOM	
	Simit	598	1.0×	**190**	1.0×	623	1.0×
	Eigen	482	0.8×	453	2.4×	354	0.6×

8.3.1 *Memory Usage.* The last two columns of Table I shows the peak total memory usage of each application as reported by the Valgrind massif tool. While Matlab enables productive experimentation, its memory usage relative to Eigen and Simit is quite large; in some cases, Matlab uses an order of magnitude more memory. Vectorizing the Matlab code can drastically increase memory usage, as in the case of the Neo-Hookean FEM example, where memory usage increased by 35×. In two applications, Simit is much more memory-efficient than Eigen due to its single representation for both the graph and matrices.

In the Elastic Shells application on the cloth, Simit uses 80% more peak memory than Eigen. The reasons for this are temporary data structures that Simit generates during neighbor index construction. These are deleted after the neighbor indices are constructed causing memory usage to drop to 333 MB during computation, which is slightly less than Eigen.

The SOFA Neo-Hookean FEM implementation uses 2.1× more memory than the Simit implementation, while the SOFA Implicit Springs uses only 60% of the memory that the Simit implementation uses. The reason for this discrepancy is differences in how SOFA represents the global system matrix used in the CG solve in these two applications. In both cases, SOFA takes advantage of the fact that the system matrix is only ever used to compute matrix-vector products to not instantiate a full sparse matrix data structure. In the Implicit Springs SOFA simply does not store any part of the system matrix, but instead computes the local blocks on demand each time the matrix is multiplied by a vector in the CG solve loop. In the Neo-Hookean FEM SOFA precomputes and stores the system matrix blocks for each element, presumably because these would be too expensive to recompute every CG iteration. When the system matrix is multiplied by a vector in the CG solve, the system matrix blocks that affect a vertex are fetched, summed, and multiplied by the relevant input vector values. Note that this strategy effectively stores the system matrix in unreduced form, that is, prior to adding the system matrix blocks that impact the same pair of vertices.

Simit, on the other hand, stores the system matrix on the vertices, that is, in reduced form. This requires more memory than not storing the matrix (as SOFA does for Implicit Springs), but less memory than storing the matrix blocks in unreduced form on the elements (as SOFA does for Neo-Hookean FEM). However, it means that no additional work has to be done to compute or sum blocks when the matrix is multiplied with a vector.

Table II. : Compilation and initialization times for Simit applications, in milliseconds. Simit code is compiled prior to the first timestep. Initialization must also be done prior to the first timestep, as well as between timesteps if arguments are rebound or the graph topology changes. Initialization time measures how long it takes to bind arguments and to create neighbor indices. The applications were run on the bunny, dragon and cloth data sets respectively with double precision floating point on an Intel Xeon E5-2695 v2 running at 2.40GHz with 128 GB of memory.

	Compilation (ms)	Initialization (ms)
Implicit Springs	37	263
Neo-Hookean FEM	52	48
Elastic Shells	38	31

8.3.2 *Compilation and Initialization.* The Simit library exposes separate compile and initialization phases. The compile phase compiles Simit programs and can be invoked after the Simit source code has been provided. The initialization phase builds indices and compiles additional harness code to bind arguments to a Simit executable function. It can be invoked after all the bindable arguments and externs have been provided. Table II shows compilation and initialization times for the three applications in Table I. Initialization times only include the time to build indices and bind arguments, and do not include the time it takes in user code to construct graphs. In all three applications, compiling and initializing Simit programs takes a fraction of a second—much less than the time it takes to execute a single frame as reported in Table I.

8.4 Simit GPU Execution

Simit programs compile to GPUs with no change to the source code. We compiled all three applications to run on an Nvidia GPU, and the

Table III. : Comparison of Simit applications running on a CPU and a GPU, and a cuSPARSE hybrid CPU-GPU implementation. The cuSPARSE FEM implementation was hand written. The applications were run on the bunny, dragon and cloth data sets respectively with single precision floating point. GPU measurements were taken on an Nvidia Titan GK210. Applications that use CG were run with 50 CG iterations per frame. With a GPU Simit achieves interactive rates on these data sets: 11 fps, 9 fps and 52 fps.

		ms per frame		Source lines	
Implicit Springs	Simit CPU	334	1.0×	93	1.0×
	Simit GPU	93	0.3×	93	1.0×
Neo-Hookean FEM	Simit CPU	668	1.0×	154	1.0×
	cuSPARSE	1420	2.1×	464	3.0×
	Simit GPU	110	0.2×	154	1.0×
Elastic Shells	Simit CPU	386	1.0×	190	1.0×
	Simit GPU	19	0.05×	190	1.0×

resulting code executed 4-20× faster than the CPU implementation. Shown in Table III is a comparison of execution times and lines of code for the single-precision implementations of each application. We saw the largest gains in the most computationally intense application, Elastic Shells, because GPUs are designed to excel at floating point operations over small dense vectors. For Elastic Shells, the runtime was improved by 20× over GPU Simit.

GPU Simit also performs faster an efficient hand-written CUDA implementation. To compare against hand-written CUDA code, the Eigen code for the Neo-Hookean FEM example was rewritten into a hybrid CPU-GPU implementation. The matrix assembly was performed using the Eigen library, using code identical to the full Eigen implementation. Using the CUDA cuSPARSE library to perform fast sparse matrix operations, the resulting matrix was then passed to the GPU to be quickly solved via the conjugate gradient method. This hybrid code approach is common for simulation developers to gain performance from a GPU by taking advantage of available libraries. At the cost of roughly 2.5× the lines of code, the CG solve step, previously half the computation time of the Eigen solution, was improved to 286ms. The GPU Simit code achieves the same speed for the CG solve step, with fewer lines of code. Due to Amdahl's Law, the hybrid implementation is limited in the speedup it can achieve, since the assembly step (which is roughly half of the runtime) is not sped up by cuSPARSE. As a result, the Simit GPU implementation still significantly outperforms this hybrid version.

8.5 Simit FLOPS and Memory Transfer

To better understand the current performance of Simit programs and to evaluate where their bottlenecks lie, we extended our compiler to instrument the generated code to count loads, stores and arithmetic operations. We used this instrumented compiler to compile the Neo-Hookean FEM application and ran it to count instructions. We validated that these numbers corresponded closely with detailed performance counters collected from a non-instrumented run on an Nvidia Titan GK210 GPU. Our instrumentation and Nvidia's counters show that Neo-Hookean FEM performs 4.3 billion integer operations and 1.2 billion floating point operations per timestep, assuming the CG performs 37 iterations. That is 57.5 GigaIOPS and 16.2 GigaFLOPS, which is far away from the peak computational throughput of this GPU. We conclude that the Simit Neo-Hookean FEM application is memory bound, which is expected for a code where most of the time is spent in a CG SpMV.

The memory bandwidth is 84.6 GigaBytes/sec, which is roughly $1/3$ of the theoretical peak bandwidth of 288.4 GigaBytes/s for this GPU. Since an SpMV indirectly loads values from the vector operand, we do not expect it to reach the peak memory bandwidth, which can typically only be approached with streaming loads. Furthermore, the simple row-parallel code emitted by the Simit compiler is less memory efficient than more sophisticated implementations that lay out memory in such a way that the GPU can coalesce loads [Weber et al. 2013], or that parallelize a row across a thread group [Bell and Garland 2008]. In the future we plan to adopt one of these more sophisticated approaches to GPU parallelization.

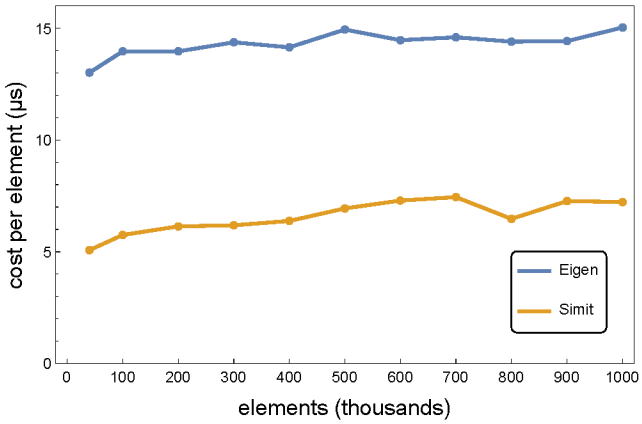


Fig. 14: Scaling performance of Eigen and Simit implementations of Neo-Hookean Tetrahedral FEM, as the number of elements in the dragon mesh are increased from approximately 100,000 elements to 1 million elements.

Table IV. : Comparison of solve times for a CG written in Simit, a CG from Eigen, and an AMG CG from AMGCL. AMGCL is an implementation of algebraic multigrid in C++ that includes a parallel implementation. The Eigen and Simit CG solvers are serial. AMGCL matrix conversion times (on average, 350ms) is excluded from these measurements. We report both the time to solution and the time per iteration of the solve. The measurements were taken on the Neo-Hookean FEM application, with convergence set to 1e-6, using the dragon data set with double precision floating point on an Intel Xeon E5-2695 v2 running at 2.40GHz with 128 GB of memory.

	To Convergence (ms)	Per CG Iteration (ms)
Simit CG	14,233	9
Eigen CG	48,693	12
AMGCL CG (Parallel)	8,474	13
AMGCL CG (Serial)	31,137	54
AMGCL CG (Parallel, BCRS)	22,099	34
AMGCL CG (Serial, BCRS)	98,738	296

8.6 Simit Scaling to Large Datasets

Figure 14 shows time per element of Eigen and Simit as the number of elements are increased in the Neo-Hookean Tetrahedral FEM application. Both implementations scale roughly linearly in the number of elements, as we expect, given that both of the major computational steps (assembly and CG with a capped maximum number of steps) scales with the number of elements times the average number of neighbors per element. Across the various sizes, Eigen averages 14.3us per element per timestep, while Simit averages 6.5us per element per timestep.

8.7 Comparison with External Solvers

In this section we compare the CG solver we implemented in Simit, and used for the experiments in Sections 8.3-8.6, with solvers developed by others. To get a direct comparison we added a solver intrinsic to Simit, and implementations of this intrinsic that use

the CG solver from the Eigen library or one of several algebraic multigrid solver variants from the AMGCL library [Demidov 2015].

We do not have a multigrid implementation in Simit, but through support for external solvers Simit applications can use this algorithm. Further, since Simit is not yet parallelized or vectorized, external solvers let applications call code that takes advantage of these hardware resources. However, because Simit optimizes and fuses linear algebra using knowledge of the blocked structure of the matrices, it is not obvious which approach yields the fastest time to solution in practice. To test this, we added code to the Simit runtime that converts matrices into standard compressed sparse row (CSR) format and calls an external solver library. We compare against both AMGCL and Eigen as the external solver, since AMGCL contains both an asymptotically better algorithm (multigrid) and a parallel implementation, while Eigen contains a standard vectorized CG used by many applications.

Table IV shows the results of replacing Simit's CG in the Neo-Hookean FEM with calls to external solvers. We exclude the time spent converting matrices from BCSR to CSR (the format used by Eigen and AMGCL), which averaged 350ms for the Dragon mesh, and run the solvers until the error is less than 10^{-6}. Compared to Simit, AMGCL converges to the answer in fewer timesteps, as we expect, though the time to solution is 2.2× slower in serial and only 53% faster in parallel. This is because Simit's cost per timestep is very low versus AMGCL. Per timestep, Simit's CG is 9ms, while AMGCL's is 13ms in parallel (54ms for serial); each timestep in AMGCL does more work due to the multigrid algorithm.

AMGCL also supports a blocked representation similar to Simit's internal BCSR representation. However, the time to solution using AMGCL's blocked backend is slower than both AMGCL's built-in (CSR) backend and Simit. This is because AMGCL's interface only accepts CSR, which is then transformed to BCSR. This shows that blocking is not the sole reason for Simit's performance, but rather the combination of native matrix assembly, which allows the compiler to emit code that directly assembles BCSR, and efficient code generated for the blocked operations. The CG implementation in Eigen is 3.4× slower than Simit, because it requires more iterations than the Simit CG to converge to a solution and because each iteration is slower.

Overall, this comparison shows that a CG implemented in Simit is efficient and can outperform hand-written external solves even though it is written in a higher-level language, due to the combination of Simit's efficient code generation and internal blocked representation. In addition, a solver written in Simit enjoys the portability that Simit provides and can be compiled to GPU code.

8.8 Comparison with a Hand-Optimized Hexahedral FEM Implementation

In this section we compare a Hexahedral Corotational FEM application written in Simit to a state-of-the-art hand-optimized implementation by Dick et al. [2011]. The Dick et al. implementation uses a geometric multigrid solver and custom problem-specific data structures to achieve impressive performance. This comparison shows the performance that can be achieved by choosing the right solver and by optimizing for discretizations with regular structure. However, the programmer must write a lot of optimized code in low level languages like C++ and CUDA to obtain this performance.

Dick and colleagues implement a hexahedral corotational finite element model, using a geometric multigrid. They exploit the regular topology of hexahedral grids to lay out data and to organize computation in a manner that avoids thread divergence, enables load coalescing on GPUs, and makes it possible to compute locations of neighbors instead of loading them from an index. The regular memory layout also makes it efficient to generate levels of the multigrid hierarchy at different resolutions. In order to compare code generated by Simit to the work of Dick et al., we implemented the Hexahedral Corotational FEM application in Simit (Section 8.1.4). However, we used a CG solver since we do not have a geometric multigrid implementation in the Simit language.

Table V. : Comparison of a Hexahedral Corotational FEM application with a CUDA implementation by Dick et al. The comparison is run with single floating-point precision. We run the comparison on a voxelized Stanford bunny with 94k and 269k elements. To account for the different solvers, we run the multigrid solver in Dick et al. for a fixed number of iterations and ensure the Simit CG runs until it achieves the same error as the multigrid solver. Both CPU and GPU timings in seconds are shown, as are the time used by Simit relative to Dick et al. For each data set and architecture, we show the total time used for each time step during simulation and time used to assemble the linear systems.

			Dick et al.	Simit	
94k	CPU	Assembly	0.16	0.79	4.8×
		Timestep	1.08	5.65	5.3×
		Lines of code	753	159	0.2×
	GPU	Assembly	0.02	0.04	1.9×
		Timestep	0.08	0.90	11.0×
		Additional code	1289	0	0.1×
269k	CPU	Assembly	0.45	2.29	5.1×
		Timestep	2.95	23.63	8.0×
	GPU	Assembly	0.05	0.09	1.8×
		Timestep	0.18	3.31	18.2×

As shown in Table V, the Simit implementation of Hexahedral Corotational FEM required 155 lines of Simit code, which includes the CG solver, and it took one person one day to implement it. We sliced out the code in Dick et al. that does the same work as the Simit program, excluding code that loads data structures and generates multigrid levels. This gave us 753 lines of C++ code for their CPU version, and an additional 637 lines of CUDA code and 652 lines of C++ to launch CUDA kernels for their GPU version—a total of 2042 lines of code. This shows the cost of writing a hand-optimized simulation application; programmers have to write a lot of low level code, and if they want to execute on a different architecture, such as GPUs, they must write additional code in something like CUDA. With Simit programmers write an order of magnitude less code using linear algebra and intuitive assembly constructs, and get portability for free.

Table V shows the benefit of using a multigrid solver and writing a hand-optimized hexahedral corotational FEM application. The implementation of Dick et al. outperforms Simit in all cases on the Stanford bunny at 94k and 269k hexahedral elements. The experiments were run with single precision floating point on an Nvidia Geforce Titan Z GPU; to account for the different solvers, we ran the multigrid solver from Dick et al. for a fixed a number of iterations and then ran Simit's CG until the same error is achieved. The main reason for their impressive performance is the use of an asymptotically superior $O(n)$ geometric multigrid solver. This solver causes their implementation to scale better as the dataset increases, which can be seen from the increase in relative performance as the dataset increases. We expect this trend would continue for larger datasets.

To get a sense of the performance impact of the Dick et al. optimizations to take advantage of the regular hexahedral discretization, we measured the time taken in assembly since this is unaffected by the difference in solver algorithms. On the GPU, Simit performs within a factor of two of the Dick et al. implementation, while

Simit's single-threaded CPU assembly is 5× slower. We believe the higher performance achieved by Dick et al. on assembly is due to two factors. First, while Simit performs assembly by scattering data from each element to its vertices, Dick et al. gathers data from each element for each vertex. This results in better memory performance on the CPU, while on the GPU it removes the need for parallel synchronization. Simit uses a scatter assembly because gather assemblies often results in redundantly computed data. However, this is not a problem for this application since every element has the same stiffness matrix. Second, Dick et al. exploits the hexahedral discretization to lay out memory in a way that improves locality and reduces GPU thread divergence.

Dick et al. have done a very good job of optimizing this application, and hand-optimized code will always have a place where peak performance is needed. However, we have a very general system that has a single general method for producing code that works reasonably well for everything our language supports. In the future, we believe we can build many of the optimizations that Dick et al. present into our compiler, due to the data independence offered by the Simit programming model (see Section 10). We also plan to add the necessary features to implement multigrid in the Simit language.

8.9 Comparison with the FreeFem++ PDE Language

In this section we compare two tetrahedral FEM codes implemented in Simit to equivalent codes implemented in the FreeFem++ PDE FEM Language. In recent years it has become popular again to develop problem-specific languages that make it very easy to develop fast applications for a class of problems. FreeFem++ is a language used in the scientific computing community for solving partial differential equations using the Finite Element Method (FEM). We implemented the Neo-Hookean FEM application from Section 8.1.2 in FreeFem++. In addition, we implemented a Laplace FEM code taken from the FreeFem++ tutorial in both Simit and Eigen.

The Laplace FEM application shows the strengths of problem-specific languages. It is very easy to implement problems that fit in the language's domain model. It takes just 8 lines of FreeFem++ code to write the Laplace FEM application and performance is decent; it runs in 2.50 seconds on the Dragon mesh. Eigen requires 125 lines of code and completes in 0.17 second, while Simit requires 61 lines of code and completes in 0.13 seconds.

The Neo-Hookean FEM application shows the limitation of problem-specific languages. Since they tend to use problem-specific abstractions you are more likely to paint yourself into a corner. That is, if you need to implement something the designers did not consider you must either find another language or contort the problem to fit. In addition, too often a problem involves more than one problem domain, and languages with too specific abstractions tend to be incompatible [Hamming 2003]. It took 681 lines of FreeFem++ code to write the Neo-Hookean FEM application and it got poor performance; one time step took 4 hours (14,425 seconds). The Eigen implementation at 364 lines of code completed in 2.3 seconds and the Simit implementation at 154 lines completed in 0.9 seconds (16,000 times faster). We contacted the FreeFem++ developers who confirmed that this is a known problem, due to the size of expressions built from their macro generation [Hecht 2015]. Although this FEM problem did not hit the FreeFem++ optimization path, we believe problem-specific language have a place and can provide good performance with very little code. However, a more general language like Simit, built around more general abstractions like graphs and matrices instead of more specific abstractions like meshes and PDE equations, is useful since it more readily generalizes to multiple domains.

9. REASONS FOR PERFORMANCE

Simit's performance comes from its design and is made possible by the careful choice of abstract data structures (hypergraphs and blocked tensors), and the choice of collection-oriented operations on these (stencils, tensor assemblies and index expressions). Little effort has so far gone into low-level performance engineering. For example, Simit does not currently emit vector instructions, nor does it optimize memory layout or run in parallel on multi-core machines, but we plan to add this in the future. Specifically, there are three keys to the performance of Simit presented in this article:

In-place Computation. The tensor assembly construct unifies the hypergraph with computations expressed as index expressions on system tensors. This lets the compiler reason about where tensors come from, and thus how their non-empty values are distributed relative to the graph. Simit uses this information to pin every value in a system tensor to a graph node or edge, which lets it schedule computation in-place on the graph. In-place computation allows efficient matrix assembly, since the graph itself becomes the sparse index structure of matrices and matrix values can be stored directly into their final location with no sparse index construction stage. Traditional sparse matrix libraries require separate insertion and compression steps; in Simit, these are replaced by the map construct, and compression disappears. It is even possible to completely remove matrix assembly by fusing it with computations that use it.

Furthermore, knowledge about matrix assembly lets the compiler optimize sparse matrix operations where the matrices have the same sparsity structure. For example, adding two sparse matrices assembled from the same graph becomes as efficient as dense vector addition. These kinds of static optimizations on sparse matrices have not been possible before, as explained by Vuduc et al.: "while sparse compilers could be used to provide the underlying implementations of sparse primitives, they do not explicitly make use of matrix structural information available, in general, only at run-time." [Vuduc et al. 2005]. The knowledge of matrix assembly provided by Simit's tensor assembly breaks the assumptions underlying this assertion and opens up the possibility of powerful static sparse optimizations.

Finally, in-place computation makes efficient parallel computation on massively parallel GPUs straightforward by assigning graph nodes and edges to parallel threads and computing using the owner-computes rule. This works well because given an assignment, Simit also knows how to parallelize system vector and matrix operations. Furthermore, these are parallelized in a way that matches the parallelization of previous stages, thereby reducing synchronization and communication. Without the in-place representation, parallel code generation would not be as effective.

Index Expression Fusion. Tensor index expressions are a powerful internal representation of computation that simplify program transformation. Index expressions are at once simpler and more general than linear algebra. Their power and uniformity makes it easy for the compiler to perform transformations like tensor operation fusion to remove tensor temporaries and the resulting memory bandwidth costs. Moreover, it can do these optimizations without the need to build in the many rules of linear algebra.

Dense Block Computation. Natively blocked tensors and hierarchical indexing not only simplify the programmer's code, they also make it trivial for the compiler to use blocked representations. Blocked matrix representations result in efficient dense or unrolled inner loops within sparse computations such as SpMV. This greatly reduces the need to look up values through an index.

10. DISCUSSION

As we argue in the introduction, the most natural way to reason about physical systems is as, simultaneously, a set of local formulas and global (configurational) operations. Programmer productivity is maximized when they are allowed to write code the way they *want* to write it: using either the local or global view of the system, without worrying about explicitly switching between the two or incurring huge performance penalties. Simit's design choices are tailored towards this goal: tensor assemblies switch between local hypergraph views and global matrix views, and index expressions together with global matrices perform global operations. Together, these operations let users write code that does not specify data layout or data access order, which greatly simplifies the code while simultaneously letting the programming system optimize for different platforms [Codd 1970]. The main alternative we considered for computation was a model with only local operators, or update functions, that apply to and update every vertex of a graph and that can read (and optionally write) to neighbors [Low et al. 2010; Pingali et al. 2011]. However, we think they are too low level, make it difficult to write complex programs, force the programmer to decide the location of certain values such as element stiffness matrices, and make it hard for compilers to optimize globally. The main alternatives to our hypergraphs with hierarchical edges were meshes and normal graphs. However, we believe meshes are too problem-specific while normal graphs require programmers to build non-binary and topological relationships into application code, which simultaneously complicates it and reduces the compiler's ability to optimize.

11. CONCLUSIONS

A key insight of this work is that the best abstraction for describing a system's structure is different from the best abstraction for describing its behavior. Sparse systems are naturally graph-structured, while behavior is often best described using linear algebra over the whole system. Simit is a new programming model that takes advantage of this duality, using tensor assembly maps to bridge between the abstractions. Using information about system sparsity, combined with operations expressed as index expressions, Simit compiles to fast code while retaining the expressibility of high-level languages like Matlab. With the ability to run programs on CPUs and GPUs, Simit attains an unprecedented level of performance portability.

We believe Simit has the potential to obtain higher performance while retaining its expressibility. So far, our implementation has only scratched the surface of what kinds of optimizations are possible with assemblies and index expressions. Future work will extend our optimization strategy for index expressions resulting from linear algebra operations. Furthermore, we have not yet implemented parallelization or vectorization of CPU code, which can provide further factors of speedup. Finally, distributed and hybrid code generation is possible given the Simit abstractions and will further improve performance.

Simit lets programmers write code at a high level and get the performance of optimized low-level code. Simit enables Matlab-like productivity with the performance of manually optimized C++ code.

ACKNOWLEDGMENTS

This material is based on work supported by the DOE awards DE-SC0005288 and DE-SC0008923, NSF DMS-1304211 and XPS-1533753, DARPA SIMPLEX, DARPA agreement FA8750-14-2-0009, and the Stanford Pervasive Parallelism Lab.

APPENDIX

A. NEO-HOOKEAN FINITE ELEMENT METHOD

We show a complete implementation of a finite element method with linear tetrahedral elements. Our implementation includes the constitutive model, the assembly stage of forces and stiffness matrices, and a linearly-implicit dynamics integrator. We implement the Neo-Hookean material model, but other material models can be plugged in by changing the stress and stress differential functions. Our assembly code defines how to compute local stiffness and forces for a single element, and Simit handles the global assembly. We also show an implementation of a linearly-implicit time integrator with a conjugate gradient linear solver. The time stepper is written in terms of linear algebra, and since it is agnostic of the underlying finite element structures it can be applied to different element types.

```
const grav = [0.0, -10.0, 0.0];

element Vert
  x : tensor[3](float);
  v : tensor[3](float);
  c : int;
  m : float;
end

element Tet
  u : float;
  l : float;
  W : float;
  B : tensor[3,3](float);
end

extern verts : set{Vert};
extern tets  : set{Tet}(verts, verts, verts, verts);

% precompute volume and shape function gradient for tets
func precomputeTetMat(inout t : Tet, v : (Vert*4))
    -> (m:tensor[verts](float))
  var M:tensor[3,3](float);
  for ii in 0:3
    for jj in 0:3
      M(jj,ii) = v(ii).x(jj) - v(3).x(jj);
    end
  end
  t.B = inv(M);
  vol = -(1.0/6.0) * det(M);
  t.W = vol;

  rho = 1000.0;
  for ii in 0:4
    m(v(ii))=0.25*rho*vol;
  end
end

export func initializeTet
  m = map precomputeTetMat to tets;
  verts.m = m;
end

% first Piola-Kirchoff stress
func PK1(u : float, l : float, F : tensor[3,3](float))
    -> (P : tensor[3,3](float))
  J = log(det(F));
  Finv = inv(F)';
  P = u*(F-Finv) + l*J*Finv;
end

% gradient of first Piola-Kirchoff stress
func dPdF(u : float, l : float, F : tensor[3,3](float),
          dF : tensor[3,3](float))
    -> (dP : tensor[3,3](float))
  J = log(det(F));
  Fi = inv(F);
```

```
  FidF = Fi*dF;
  dP = u*dF + (u-l*J) * Fi' * FidF' + l*trace(FidF)*Fi';
end

% assemble lumped mass matrix and gravitional force
func compute_mass(v : Vert)
    -> (M : tensor[verts, verts](tensor[3,3](float)),
        fg : tensor[verts](tensor[3](float)))
  M(v,v) = v.m * I;
  if(v.c <= 0)
    fg(v) = v.m * grav;
  end
end

% assemble internal forces, fixed vertices contribute
% no force
func compute_force(e : Tet, v : (Vert*4))
    -> (f : tensor[verts](tensor[3](float)))
  var Ds : tensor[3,3](float);
  for ii in 0:3
    for jj in 0:3
      Ds(jj,ii) = v(ii).x(jj) - v(3).x(jj);
    end
  end
  F = Ds*e.B;

  P = PK1(e.u, e.l, F);
  H = -e.W * P * e.B';

  for ii in 0:3
    fi = H(:,ii);

    if (v(ii).c <= 0)
      f(v(ii)) = fi ;
    end

    if (v(3).c <= 0)
      f(v(3)) = -fi;
    end
  end
end

% assemble stiffness matrix, fixed vertices contribute
% nothing to stiffness matrix
func compute_stiffness(e : Tet, v : (Vert*4))
    -> (K : tensor[verts,verts](tensor[3,3](float)))
  var Ds : tensor[3,3](float);
  var dFRow : tensor[4,3](float);
  m = 0.01;

  for ii in 0:3
    for jj in 0:3
      Ds(jj,ii) = v(ii).x(jj)-v(3).x(jj);
    end
  end

  F = Ds*e.B;
  for ii in 0:3
    for ll in 0:3
      dFRow(ii,ll) = e.B(ii,ll);
    end
    dFRow(3, ii) = -(e.B(0, ii)+e.B(1, ii)+e.B(2, ii));
  end

  for row in 0:4
    var Kb : tensor[4,3,3](float) = 0.0;
    for kk in 0:3
      var dF : tensor[3,3](float) = 0.0;
      for ll in 0:3
        dF(kk, ll) = dFRow(row, ll);
      end
      dP = dPdF(e.u, e.l, F, dF);
      dH = -e.W * dP * e.B';

      for ii in 0:3
        for ll in 0:3
          Kb(ii,ll, kk) = dH(ll, ii);
        end
        Kb(3, ii, kk) = -(dH(ii, 0)+dH(ii, 1)+dH(ii, 2));
      end
    end

    for jj in 0:4
      if(v(jj).c <= 0) and (v(row).c <= 0)
        K(v(jj), v(row)) = Kb(:,:,jj);
      end
    end
  end
end

% conjugate gradient with no preconditioning.
func CG(A : tensor[verts,verts](tensor[3,3](float)),
        b : tensor[verts](tensor[3](float)),
        x0 : tensor[verts](tensor[3](float)),
        tol : float, maxIter : int)
    -> (x : tensor[verts](tensor[3](float)))
  r = b - (A*x0);
  p = r;
  iter = 0;
  x = x0;
  normr2 = dot(r, r);
  while (normr2 > tol) and (iter < maxiters)
    Ap = A * p;
    denom = dot(p, Ap);
    alpha = normr2 / denom;
    x = x + alpha * p;
    normr2old = normr2;
    r = r - alpha * Ap;
    normr2 = dot(r, r);
    beta = normr2 / normr2old;
    p = r + beta * p;
    iter = iter + 1;
  end
end

% linearly-implicit time stepper with CG solver
export func main
  h = 0.01;
  tol = 1e-12;
  maxiters = 50;

  M,fg = map compute_mass to verts reduce +;
  f = map compute_force to tets reduce +;
  K = map compute_stiffness to tets reduce +;

  A = M - (h*h) * K;
  b = M*verts.v + h*(f+fg);

  x0 = verts.v;
  verts.v = CG(A, b, x0, tol, maxIter);
  verts.x = h * verts.v + verts.x;
end
```

REFERENCES

Bruce G. Baumgart. 1972. *Winged Edge Polyhedron Representation*. Technical Report. Stanford University.

Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 356. MIT press Cambridge.

David Blythe. 2006. The Direct3D 10 system. *ACM Transactions on Graphics* 25, 3 (July 2006), 724–734.

Mario Botsch, David Bommes, and Leif Kobbelt. 2005. Efficient linear system solvers for mesh processing. In *Mathematics of Surfaces XI*. Springer, 62–83.

M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. 2002. OpenMesh – a generic and efficient polygon mesh data structure. (2002).

CGAL. 2015. Computational Geometry Algorithms Library. http://www.cgal.org. (2015). Accessed: 2015-09-24.

Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

AB Comsol. 2005. COMSOL multiphysics users guide. *Version: September* (2005).

Erwin Coumans and others. 2006. Bullet physics library. *Open source: bulletphysics. org* 4, 6 (2006).

Denis Demidov. 2015. AMGCL. http://ddemidov.github.io/amgcl. (2015). Accessed: 2015-09-24.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 9, 12 pages.

Christian Dick, Joachim Georgii, and Rüdiger Westermann. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory* 19, 2 (2011), 801–816.

Pradeep Dubey, Pat Hanrahan, Ronald Fedkiw, Michael Lentine, and Craig Schroeder. 2011. PhysBAM: Physically Based Simulation. In *ACM SIGGRAPH 2011 Courses (SIGGRAPH '11)*. ACM, New York, NY, USA, Article 10, 22 pages.

C.M. Eastman and S.F. Weiss. 1982. Tree structures for high dimensionality nearest neighbor searching. *Information Systems* 7, 2 (1982), 115–122.

Albert. Einstein. 1916. The Foundation of the General Theory of Relativity. *Annalen der Physik* 354 (1916), 769–822.

C. Elliott. 2001. Functional Image Synthesis. In *Proceedings of Bridges*.

François Faure, Jérémie Allard, Stéphane Cotin, Paul Neumann, Pierre-Jean Bensoussan, Christian Duriez, Hervé Delingette, and Laurent Grisoni. 2007. SOFA: A modular yet efficient simulation framework. In *Surgetica 2007 - Computer-Aided Medical Interventions: tools and applications (Surgetica 2007, Gestes médicaux chirurgicaux assistés par ordinateur)*, Philippe Merloz and Jocelyne Troccaz (Eds.). Chambéry, France, 101–108.

Eitan Grinspun, Anil N. Hirani, Mathieu Desbrun, and Peter Schröder. 2003. Discrete Shells. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '03)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 62–67. http://dl.acm.org/citation.cfm?id=846276.846284

Gaël Guennebaud, Benoît Jacob, and others. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

Brian Guenter and Sung-Hee Lee. 2009. *Symbolic Dynamics and Geometry*. AK Peters.

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Trans. Graph.* 4, 2 (1985), 74–123.

Richard Hamming. 2003. History of Computer—Software. In *Art of Doing Science and Engineering: Learning to Learn*. CRC Press.

Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*. 289–298.

F. Hecht. 2015. private communication. (May 2015).

Hibbett, Karlsson, and Sorensen. 1998. *ABAQUS/standard: User's Manual*. Vol. 1. Hibbitt, Karlsson & Sorensen.

G. Holzmann. 1988. *Beyond Photography*. Prentice Hall.

Monica S. Lam Jiwon Seo, Stephen Guo. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *IEEE 29th International Conference on Data Engineering*.

David R. Kincaid, Thomas C. Oppe, and David M. Young. 1989. *ITPACKV 2D User's Guide*.

Peter Kohnke. 1999. *ANSYS theory reference*. Ansys.

Karen Liu. 2014. Dynamic Animation and Robotics Toolkit. (2014).

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence*.

William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics* 22, 3 (July 2003), 896–907.

MATLAB. 2014. *version 8.3.0 (R2014a)*. The MathWorks Inc., Natick, Massachusetts.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011. Efficient elasticity for character skinning with contact and collisions. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 37.

M Mooney. 1940. A theory of large elastic deformation. *Journal of applied physics* 11, 9 (1940), 582–592.

Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and others. 2010. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 66.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25.

J Pommier and Y Renard. 2005. Getfem++, an open source generic C++ library for finite element methods. (2005).

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. DOI:http://dx.doi.org/10.1145/2185520.2185528

Gregorio Ricci-Curbastro and Tullio Levi-Civita. 1901. Mthodes de calcul diffrentiel absolu et leurs applications. *Math. Ann.* 54 (1901), 125–201. http://eudml.org/doc/157997

Mark Segal and Kurt Akeley. 1994. *The OpenGL graphics system: a specification*.

Fun Shing Sin, Daniel Schroeder, and J Barbič. 2013. Vega: Non-Linear FEM Deformable Object Simulator. In *Computer Graphics Forum*, Vol. 32. 36–48.

Russell Smith and others. 2005. Open dynamics engine. (2005).

Eric Sedlar Sungpack Hong, Hassan Chafi and Kunle Olukotun. 2012. GreenMarl: A DSL for Easy and Efficient Graph Analysis. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. 2013. Openfab: A programmable pipeline for multi-material fabrication. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 136.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, Vol. 16. 521–530. DOI:http://dx.doi.org/10.1088/1742-6596/16/1/071

Daniel Weber, Jan Bender, Markus Schnoes, Andre Stork, and Dieter Fellner. 2013. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum* 32, 1 (2013).

Received May 22; accepted August 27